最新IT书籍PDF

Neo4j权威指南

图数据库——大数据时代的新利器

Neo4j官方推荐

清华大学出版社

写作团队

Neo4j权威指南

推荐序

CCTV《超越》栏目：关于图数据库的梦想

本书内容

本书技术支持

创作团队和致谢

前言

目 录

第 1 章 Neo4j 图数据库基础 1

1.1 图数据库的产生背景 2

1.2 图数据库基础 7

1.2.1 图数据库介绍 7

1.2.2 图数据模型 7

1.2.3 图计算引擎 8

1.2.4 图数据库的历史 9

1.3 图数据库与关系数据库的对比 10

1.3.1 关系数据库的弊端 10

1.3.2 图数据模型的优势 10

1.4 图数据库与其他 NoSQL 数据库的对比 12

1.4.1 其他 NoSQL 数据库的弊端 12

1.4.2 将键值对存储与图数据库相关联 12

1.4.3 将文档存储与图数据库相关联 13

1.5 Neo4j 概述 13

1.6 Neo4j 的体系结构 15

1.6.1 免索引邻接 16

1.6.2 Neo4j 底层存储结构 17

1.6.3 Neo4j 的遍历方式 21

1.6.4 Neo4j 的存储优化 22

第 2 章 Neo4j 基础入门 24

2.1 Neo4j 的安装部署 24

2.1.1 Neo4j 安装包的下载 24

2.1.2 在各个操作系统上的安装 26

2.1.3 Neo4j 的启动 32

2.2 Neo4j 管理平台的使用 33

2.2.1 Web 管理平台的使用 33

2.2.2 neo4j-shell 的使用 37
2.3 Neo4j 图数据中基本元素与概念 41
2.3.1 节点 41
2.3.2 关系 41
2.3.3 属性 42
2.3.4 路径 42
2.3.5 遍历（Traversal） 43
2.4 官方入门实例介绍 43
2.4.1 创建图数据 44
2.4.2 检索节点 46
2.4.3 查询关系 49
2.4.4 思考与练习 52
2.4.5 清空数据库 54
2.5 批量导入工具的使用 54
2.5.1 获取 CSV 文件 54
2.5.2 使用 Load CSV 指令导入到 Neo4j 58
2.5.3 使用 neo4j-import 工具导入到 Neo4j 61

第 3 章 Neo4j 之 Cypher 63

3.1 Cypher 概述 63
3.1.1 Cypher 是什么 63
3.1.2 模式（Patterns） 65
3.1.3 查询和更新图 66
3.1.4 事务 67
3.1.5 唯一性 68
3.1.6 兼容性 69
3.2 基本语法 70
3.2.1 类型 70
3.2.2 表达式 70
3.2.3 变量 73
3.2.4 参数 73
3.2.5 运算符 77
3.2.6 注释 78
3.2.7 模式（Patterns） 78

3.2.8 列表 81
3.2.9 空值 87
3.3 语句 89
3.3.1 MATCH 89
3.3.2 OPTINAL MATCH 101
3.3.3 WHERE 102
3.3.4 START 113
3.3.5 Aggregation 114
3.3.6 LOAD CSV 121
3.3.7 CREATE 125
3.3.8 MERGE 129
3.3.9 SET 138
3.3.10 DELETE 142
3.3.11 REMOVE 143
3.3.12 FOREACH 145
3.3.13 CREATE UNIQUE 146
3.3.14 RETURN 149
3.3.15 ORDER BY 153
3.3.16 LIMIT 155
3.3.17 SKIP 156
3.3.18 WITH 158
3.3.19 UNWIND 160
3.3.20 UNION 161
3.3.21 CALL 163
3.4 函数 167
3.4.1 断言（Predicate）函数 167
3.4.2 标量（Scalar）函数 170
3.4.3 列表（List）函数 177
3.4.4 数学函数 182
3.4.5 字符串函数 193
3.4.6 自定义函数 199
3.5 模式（Schema） 200
3.5.1 索引 200
3.5.2 约束 203

3.5.3 统计 209
3.6 查询调优 210
3.6.1 查询如何执行 210
3.6.2 查询性能分析 211
3.6.3 查询调优举例 211
3.6.4 USING 213
3.7 执行计划 220
3.7.1 开始点运算符 221
3.7.2 Expand 运算符 224
3.7.3 组合运算符 226
3.7.4 行运算符 234
3.7.5 更新运算符 241
3.7.6 最短路径规划 243

第 4 章 Neo4j 程序开发 246

4.1 Neo4j 开发入门 246
4.1.1 Java 嵌入式开发模式 246
4.1.2 各语言驱动包开发模式 246
4.2 Java API 嵌入式开发模式 247
4.2.1 Java 开发前的准备工作 248
4.2.2 创建 Neo4j 图实例 253
4.2.3 图数据遍历功能 257
4.2.4 数据索引 264
4.2.5 过程 273
4.2.6 事务管理 280
4.2.7 使用 Java 在线备份 Neo4j 286
4.2.8 使用 JMX 监控 Neo4j 286
4.3 各语言驱动包开发模式 288
4.3.1 驱动开发入门 289
4.3.2 配置和连接 292
4.3.3 执行 Cypher 语句 297
4.3.4 返回结果 301
4.3.5 数据类型 308
4.3.6 异常 310

4.3.7 其他语言驱动 311
4.4 Neo4j HTTP API 312
4.4.1 流 312
4.4.2 认证和授权 312
4.4.3 在一个请求中使用事务 316
4.4.4 执行多条语句 317
4.4.5 运行一个事务 318
4.4.6 在打开的事务中执行语句 319
4.4.7 重置超时事务 319
4.4.8 提交事务 320
4.4.9 回滚事务 321
4.4.10 查询统计信息 321
4.4.11 以图格式返回结果 322
4.4.12 错误处理 324
4.4.13 在事务中处理错误 325
4.5 其他开发技术介绍 326
4.5.1 Spring-Data-Neo4j 326
4.5.2 Spring-Data-Neo4 在项目中的部署 326
4.5.3 使用 Neo4j-OGM 的对象图映射 327
4.5.4 使用 JDBC 连接 Neo4j 328
4.5.5 JCypher 329
4.5.6 Groovy＆Grails：Neo4j Grails 插件 329
4.5.7 Clojure：Neocons 329
4.5.8 Scala：AnormCypher 330
4.5.9 JPA：Hibernate OGM 330

第 5 章 Neo4j 数据库管理 331

5.1 部署 331
5.1.1 系统需求 331
5.1.2 文件位置 332
5.1.3 重要端口 333
5.1.4 设置初始密码 334
5.1.5 等待 Neo4j 启动 334
5.1.6 使用数据收集器 335

5.1.7 配置 Neo4j 连接器 336
5.1.8 安装证书 337
5.2 监控 338
5.2.1 指标 340
5.2.2 日志 344
5.2.3 查询管理 346
5.2.4 因果集群监控相关方法 348
5.3 安全管理 350
5.3.1 Neo4j 社区版用户管理 350
5.3.2 认证和授权 352
5.3.3 与 LDAP 集成 362
5.3.4 子图访问控制 366
5.3.5 安全清单 367
5.4 Neo4j 数据库运维与优化 368
5.4.1 内存调优 368
5.4.2 事务日志 371
5.4.3 压缩存储 371
5.4.4 Linux 文件系统调优 373
5.4.5 磁盘、内存及相关提示 373
5.5 Neo4j 数据库的备份与恢复 374
5.5.1 备份简介 374
5.5.2 执行备份 375
5.5.3 恢复备份 377
5.6 Neo4j 数据库管理相关工具 378
5.6.1 导入工具 378
5.6.2 Cypher Shell 382
5.6.3 Neo4j 数据转储和加载 384
5.6.4 一致性检查工具 385

第 6 章 Neo4j 集群技术 387

6.1 因果集群 387
6.1.1 初识因果集群 388
6.1.2 操作视图 388
6.1.3 应用视图 389

6.1.4 因果集群的生命周期 390
6.1.5 创建一个新的因果集群 395
6.1.6 数据中心灾难恢复 398
6.1.7 因果集群的数据迁移和恢复 400
6.1.8 因果集群的其他设置 401
6.2 高可用性集群 401
6.2.1 高可用性集群概述 402
6.2.2 建立 Neo4j 高可用性集群 404
6.2.3 状态信息端点 406
6.2.4 HAProxy 用于负载均衡 408
6.3 集群的备份与恢复 410
6.4 Neo4j 单实例和高可用性集群版本升级 411
6.4.1 单实例升级 411
6.4.2 从高可用性集群升级 412

第 7 章 Neo4j 应用案例 413

7.1 应用案例概述 413
7.2 欺诈检测 416
7.2.1 第一方银行欺诈 417
7.2.2 保险欺诈 424
7.2.3 电子商务欺诈 427
7.2.4 小结 428
7.3 科研导图 428
7.4 电子邮件监测 437
7.5 工商企业图谱 442
7.6 社交网络 449

第 8 章 Neo4j 高级应用 458

8.1 高级索引 458
8.1.1 空间索引（Neo4j Spatial） 458
8.1.2 自定义中文全文索引 468
8.2 在 Docker 环境下部署 Neo4j 471
8.2.1 概览 472
8.2.2 Docker 配置 472

8.2.3 Neo4j 配置 473
8.3 自定义批量导入 476
8.4 Neo4j 与图计算 479
8.4.1 Neo4j-Spark-Connector 480
8.4.2 Neo4j-Spark-Connector 提供的 API 482
8.5 Neo4j 与自然语言处理 483
8.5.1 计算聚合相关性 484
8.5.2 将文本数据建模为邻接图 484
8.5.3 加载数据 486
8.5.4 挖掘单词之间的关系 488
8.6 其他高级应用 490

第 9 章 Neo4j 简体中文版 491

9.1 简介 491
9.2 下载及安装 496
9.3 启动及浏览 497
9.4 指定节点图片 499
9.5 指定节点尺寸 501
9.6 指定节点颜色 502
9.7 指定关系粗细 503
9.8 指定关系颜色 504
9.9 智能查询 505
9.10 导入精灵 508
9.10.1 安装导入精灵 508
9.10.2 启动导入精灵 509
9.10.3 准备 Excel 样本文件 510
9.10.4 导入 Excel 样本文件 511
9.10.5 准备 MySQL 样本数据库 513
9.10.6 导入 MySQL 样本数据库 515
9.10.7 查看导入结果 516
9.10.8 导入精灵映射机制 517

附录 A Neo4j 配置设置 521

附录 B Neo4j 内建过程 550

第 1 章

◂ Neo4j 图数据库基础 ▸

本章包括以下内容：

- 图数据库的产生背景
- 图数据库基础
- 图数据库与关系数据库的对比
- 图数据库与其他 NoSQL 数据库的对比
- Neo4j 概述
- Neo4j 体系结构图解

大多数读者，即便是计算机科班出身，如果没有紧跟数据库发展，一听到 Neo4j 定然会比较陌生，但一般来说肯定会或多或少地接触到那些耳熟能详的传统关系型数据库产品，比如：Oracle、MySQL、SQL Server 等。事实上，Neo4j 是一种新型的数据库产品，除了能像传统关系型数据库支持存储、分析、处理数据的功能以外，它以数学中的图论为理论根基，更擅长海量数据之间的复杂关系分析，因此，在学术界和产业界通常将它称作：图数据库（Graph Database，广义上属于 NoSQL 数据库的一种）。对于广大使用者而言，如果不需要深入数据库本身的研发，Neo4j 与传统关系型数据库相似，不需要系统学习较为高深的建立在数学集合论基础之上的关系代数，仅需掌握一定的 SQL（Structured Query Language，结构化查询语言）知识。同样，使用 Neo4j 不需要钻研深奥的图论的理论知识，只需了解与 SQL 语言类似的 Cypher 语言即可。作为最近几年才发展起来的全新数据库技术，Neo4j 紧随大数据时代的步伐不断前行，越发凸显出相对于传统关系型数据库的强大优势，必将成为这个时代一颗璀璨的明珠！本书将带您一起遨游于 Neo4j 图数据库这一极富迷人魅力的广阔海洋！

本章作为全书的第 1 章，主要有两个目标：了解图数据库的基础知识和认识 Neo4j。首先介绍图数据的基本知识并将图数据库与传统数据库进行对比分析；然后介绍 Neo4j 图数据库的产生、发展以及其性能优势；最后一节介绍 Neo4j 的体系结构，这样可以对 Neo4j 的技术架构有个初步的认识。

本章将只讨论理论、概念性的知识，有了一定的理论基础后就可以在后续章节中更好地学习 Neo4j 的技术知识。

1.1 图数据库的产生背景

任何一项重大科学技术的产生都有其历史背景和必然规律。同样，作为计算机科学基础软件领域的数据库技术也有其产生和发展的历史必然性。下面简要回顾数据库技术发展历程中具有里程碑意义的事件。

1. 数据库技术发展的里程碑事件

（1）1962 年，数据库（Database）一词最早流行于加利福尼亚州（硅谷所在地）一些系统研发公司的技术备忘录中。

（2）1968 年，伴随阿波罗登月计划，商业数据库雏形诞生，IBM 公司的 IMS（Information Management System）、Mainframe，以及 Navigational 数据库技术出现。

（3）1969 年，美国国防部召开的数据系统语言会议（Conference on Data Systems Languages，CODASYL）发布了一份“DBTG（Database Task Group）报告”，标志网状数据库系统进入标准化进程。

（4）1970 年，IBM 公司研究员 Edgar Frank Codd 发表了题为“A Relational Model of Data for Large Shared Data Banks”（译为“大型共享数据库的数据关系模型”）的论文，这篇论文提出了关系模型的概念，奠定了关系模型的理论基础，他本人被誉为“关系数据库之父”，并于 1981 年获得了有“计算机界的诺贝尔奖”之称的“图灵奖”。

（5）1974 年，IBM 公司的校企联合计划中，与加州大学伯克利分校 Ingres 数据库研究项目携手创建了 RDBMS（Relation DataBase Management System，关系数据库管理系统）System R 原型。

（6）1979 年，因 IBM 公司战略调整其为当时主流的层次数据库，并剥离出处于萌芽状态的关系数据库；加州大学伯克利分校 Ingres 数据库研究项目联合 Oracle 公司的 Larry Ellison 创建了第一个商业关系数据库产品。

（7）1983 年，IBM 公司发布其第一款自主研发的关系数据库产品 DB2。

（8）1984 年，天睿公司（Terodata Corporation）发布第一个 MPP（Massive Parallel Processor，大规模并行处理）分布式数据库专用平台，或称为无共享架构（Sharing Nothing Architecture），有效提升了系统整体性能。

（9）1986 年，首款面向对象数据库 GemStone/S 出现。

（10）1988 年，IBM 公司研究员率先提出数据仓库（Data Warehouse），主要用于复杂数据分析，并制定相关行业标准，将数据处理分为两大块业务：联机事务处理（On-Line Transaction Processing，OLTP）和联机分析处理（On-Line Analytical Processing，OLAP）。

（11）1995 年，瑞典 MySQL AB 公司发布第一款开源关系数据库 MySQL。

（12）1996 年，第一款对象关系数据库 Illustra 问世，支持高效的对象和关系数据存储与管理。

（13）1998 年，随着互联网的兴起，Lawrence Edward Page 和 Sergey Brin 在斯坦福大学宿舍内共同开发了谷歌在线搜索引擎。

（14）1998 年，James Gray 因在数据库和事务处理方面的突出贡献而获 1998 年的图灵奖。

（15）2003 年，MarkLogic 公司发布第一款 NoSQL（Not Only SQL，泛指非关系型的数据库）数据库解决方案，XML 数据库。

（16）2005 年，受 Google 公司的 Map/Reduce 和 Google File System（GFS）的启发，由 Apache 基金会所开发的分布式系统基础架构 Hadoop 发布。

（17）2005 年，针对普遍存在的流数据，Streambase 公司发布第一款支持流数据处理的复杂事件处理（Complex Event Processing，CEP）解决方案。

（18）2007 年，Neo4j 公司推出第一款商用 NoSQL 图数据库，高效支持数据之间的复杂关系分析。

（19）2009 年，分布式文件存储的数据库 MongoDB 发布，为 Web 应用提供可扩展的高性能数据存储解决方案。

（20）2010 年，HBase 发行，采用列式存储而非行式，在 Hadoop 之上提供类似 Bigtable 能力，支持非结构化数据存储。

（21）2013 年，有媒体将 2013 年称为“大数据元年”，“数据即资源”受到广泛认可。

（22）2014 年，Michael Stonebraker 因对现代数据库的概念和实践作出的根本性贡献获得 2014 年图灵奖。

（23）2015 年，大数据、云计算技术得到广大企业和各国政府的青睐，技术和平台上突飞猛进，比如 VoltDB、Spark、Drill 等，单是 Apache 基金会就发布超过 25 个数据工程项目。

2. 数据的 4V

不难发现，数据库技术已经有 50 多年的历史，在计算机领域中算得上是一直保持蓬勃生机与活力的技术之一。其主要表现在如下几个方面。

（1）数据种类多样化。数据从最初的结构化数据（如数字、符号等信息）拓展为半结构化数据（如 HTML 文档等），再到非结构化数据（如办公文档、文本、图片、XML、各类报表、图像和音视频信息等）。IDC（国际数据公司）的调查报告显示：企业 80%的数据都是非结构化数据，且每年按指数增长 60%。

（2）数据量增长迅猛。根据 IDC 的监测统计，2011 年全球数据总量已经达到 1.8 ZB（2^{70} 字节），而这个数值还在以每 2 年翻一番的速度增长，预计到 2020 年，全球将总共拥有 35 ZB 的数据量，比 2011 年增长近 20 倍。也就是说，近 2 年产生的数据总量相当于人类有史以来所有数据量的总和。

（3）数据产生速度快。受摩尔定律的支配，计算机硬件及网络速度每 18 个月增加一倍，显然导致数据产生的速度飞速提升。

（4）数据价值备受关注。数据价值由先前的少量局部数据内的价值转向大量全局数据之间的价值，不断涌现出数据仓库、数据挖掘、分布式数据分析等技术。

上述 4 个方面，无不集中展现到当前所处的大数据时代这一历史舞台中，凸显出数据的 4V（Volume 大量、Velocity 高速、Variety 多样、Value 价值）特性。从上述数据库的简要发展历程稍作分析，便不难发现：数据信息化进程和对数据价值的不断渴求可以说是持续支撑数据库技术发展的不竭动力源泉。

3. 关系型数据库（SQL 类）和非关系型数据库（NoSQL 类）

在这精彩纷呈的数据库技术发展历程中，是否存在一种“包打天下的武学秘籍，包治百病的旷世神方”呢？它能有效解决上述所有难题，作为数据库领域霸主的关系型数据库行吗？或许这个问题似乎难以回答。将数据库技术发展历程中的主要技术稍加归类整理，可绘制出如图 1-1 所示的数据库分类图。

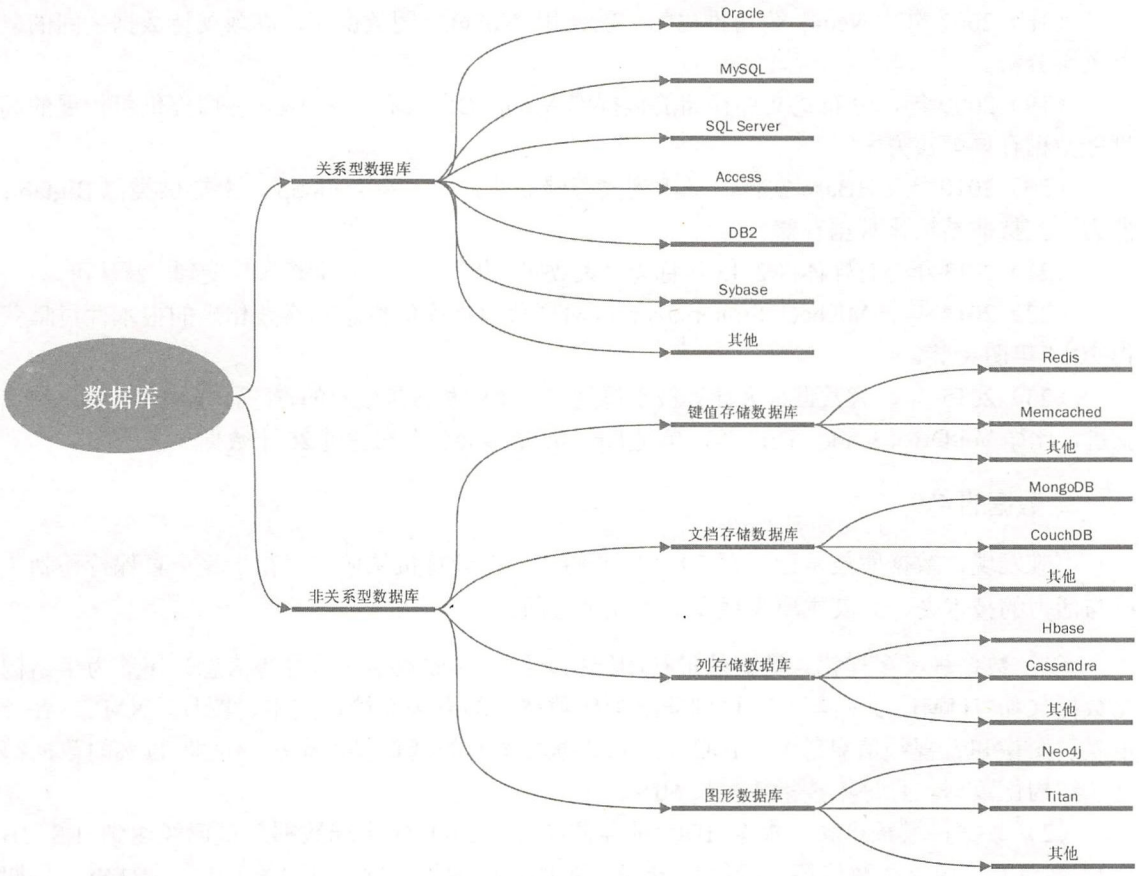


图 1-1　数据库分类图

上图非常形象地将数据库技术分为两大类：关系型数据库（SQL 类）和非关系型数据库（NoSQL 类），关系型数据库源源不断地为 NoSQL 类数据库提供理论依据，也就是说，NoSQL 类数据库的发展是以关系型数据库为基础，很多思想和智慧都来源于关系型数据库，比如：事务、分布处理、集群、查询语言等。但 NoSQL 数据库相对于霸主地位的关系型数据库而言，无疑是一种全新的思维和创新。为进一步分析上述枝繁叶茂的 NoSQL 数据库，有必要对其进行分类。现有的分类方法多种多样，各方法之间分出来的类或子类还可能存在重叠，主流的方法[1]（Mikayel Vardanyan 依据数据模型的不同给出的一个基本分类）分为四大类。

[1] http://www.monitis.com/blog/picking-the-right-nosql-database-tool/

（1）键值存储（Key-Value）数据库

主要采用哈希表技术，存储特定的键和指向特定的数据指针。该模型简单、易于部署。例如：Redis、Memcached、Riak KV、Hazelcast、Ehcache、Voldemort、Oracle BDB 等。

（2）文档型数据库

以嵌入式版本化文档为数据模型，其灵感是来自于 Lotus Notes 办公软件，支持全文检索、关键字查询等功能。例如：MongoDB、Amazon DynamoDB、Couchbase、CouchDB、SequoiaDB 等。

（3）列存储数据库

列存储数据库是指数据存储采用列式存储架构，相比传统的行式存储架构，数据访问速度更快，压缩率更高，支持大规模横向扩展。例如：Cassandra、HBase、Riak、GBase 8a 等。

（4）图（Graph）数据库

以图论为理论根基，用节点和关系所组成的图为真实世界直观建模，支持百亿乃至千亿量级规模的巨型图的高效关系运算和复杂关系分析，例如：Neo4j、OrientDB、Titan 等。

上述四类 NoSQL 数据库依据其数据模型的不同，均表现出各自的优势和劣势，以及适应的典型应用场景。具体归类如表 1-1 所示。

表 1-1　NoSQL 数据库的四大分类分析

分类	数据模型	优势	劣势	典型应用场景
键值数据库	哈希表	查找速度快	数据无结构化，通常只被当作字符串或者二进制数据	内容缓存，主要用于处理大量数据的高访问负载，也用于一些日志系统等
列存储数据库	列式数据存储架构	查找速度快，支持分布横向扩展，数据压缩率高	功能相对受限	分布式文件系统
文档型数据库	键值对扩展	数据结构要求不严格，表结构可变，不需要预先定义表结构	查询性能不高，缺乏统一的查询语法	Web 应用
图数据库	节点和关系组成的图	利用图结构相关算法。比如最短路径，节点度关系查找等	可能需要对整个图做计算，不利于图数据分布存储	社交网络、推荐系统、意向图、消费图、兴趣图、关系图谱等

另一方面，大量的研发人员或团队对五类主流数据库产品（键值数据库、列存储数据库、文档型数据库、图数据库、关系数据库）进行过大量的比较和分析，比较的指标涵盖：性能、可扩展性、复杂性等多个维度。表 1-2 列出了一个典型的分析结果，它是 Ben Scofield 在 YCSB 开源数据库基准测试（Yahoo! Cloud Serving Benchmark）上所测试分析的结果[2]。

[2] http://www.slideshare.net/bscofield/nosql-codemash-2010

表 1-2　各类数据库主要指标分析

分类	性能	可扩展性	灵活性	复杂性	功能性
键值数据库	高	高	高	无	可变（无）
列存储数据库	高	高	一般	低	很少
文档型数据库	高	可变（高）	高	低	可变（低）
图数据库	可变	可变	高	高	图论
关系数据库	可变	可变	低	一般	关系代数

通过上述较为系统全面的分析，无论是从宏观的优劣势和典型应用场景方面，还是微观具体的基准测试方面，之前的问题有了更为清晰的答案。正如数据库领域的布道者、图灵奖得主 Michael Stonebraker 对数据库未来发展所预言的那样 One size will fit none（单一模式不能包打天下），也就是告诫我们需要将更多的精力花在选择最合适的技术去有针对性地高效解决所面临的具体问题，恰恰回归到“具体问题具体分析”这一普适的哲学原理。

4. 图数据库的发展

具体到图数据库而言，相对于传统关系数据库这一“老大哥”就最多只能称作“小老弟”。作为图数据库的领跑者——Neo4j 最初发布版本是在 2007 年，一路奔跑前行至今也不过 10 年。可见这一新型数据库技术有其固有的、旺盛的生命力，细想不难发现，这来源于：图是一种能对现实世界进行普适直观的建模工具，通俗地说，图无处不在！我们信手就可拈来一大批身边与吃穿住行生活密切相关的图事例，比如人际交往、购物消费、交通网络、旅游规划等。

正如全球著名的信息咨询公司——高德纳咨询公司（Gartner Group）所分析[3]：商业世界中有 5 个非常有价值的图：Social Graph（社交图）、Intent Graph（意向图）、Consumption Graph（消费图）、Interest Graph（兴趣图）、Mobile Graph（移动图），并指出应用这些图的能力是一个“可持续的竞争优势”。

图数据库在数据库领域如此“晚产”，正如其理论基础图论在数学界一样像“新生儿”，图论最早可追溯到图论创始人瑞典数学家欧拉（Leornhard Euler）于 1738 年解决了柯尼斯堡七桥问题。现在图论应用极其广泛，但大发展时期是 20 世纪 40~60 年代，一系列图论相关研究取得了突破性进展，包括：拟阵理论、超图理论、极图理论、代数图论、拓扑图论等。

从之前罗列的数据库领域重要事件和表 1-1 中所示的内容，细心的读者不难发现：图数据库归类到网状数据库之列，从数据库模型上主要有三大类：层次型、网状型、关系型，网状数据库标准化的出现还早于关系数据模型的提出，而为什么关系型数据库却后来居上，长期占住数据库领域的霸主地位呢？要回答这个问题，其实不难，稍对计算机发展历史有一定了解的人就能较容易地找到答案。从表 1-2 中单看图数据库和关系数据库两行就不难发现，图数据库的实现复杂性远高于关系数据库；另一方面，再回溯上世纪，回看那时的硬件性能无法跟现在相提并论，单说处理器速度和内存容量，要实现这一令人神往又如此复杂的图数据库谈何容易，完全可以说是一种“奢求”！因此，当时的研发人员只能另辟蹊径，在数据模型上做文章，降低建模的复杂度，采用二维表（在关系数据库中称为“关系”）这一更易

[3] http://www.gartner.com/id=2081316

于理解和实现的创新性技术来构建数据库。

至此，关系数据库中的“关系”二字，还真是与我们所熟知的，比如人际关系、借贷关系等完全不搭界，仅仅是“关系数据库之父”Edgar Frank Codd 在其论文中将：在集合论基础上构建二维表称作关系罢了。那是不是说传统关系型数据库不能存储、管理我们所熟知的这些关系呢？这个显然不是，而是将这些熟知的关系隐藏转化为以集合论为基础的关系代数上的一种常用的连接操作（Join），最终在当时用关系模型的易理解性和理论可行性来超越网状模型之上的图数据库的复杂性，成就了关系型数据库的持久辉煌！

诚然，得益于信息技术的飞速发展，无论是硬件、网络，还是软件等各方面都今非昔比，图数据库自从 2007 年第一个可商用 Neo4j 产品的出现，就成为政界、学界、商界、媒界持续关注的焦点，其发展势头绝不亚于当年的关系数据库，俨然成为能高效、便捷、直观地处理海量、快速、多样的数据，并能快速挖掘其中纷繁复杂关系的一把利器！

1.2 图数据库基础

1.2.1 图数据库介绍

图数据库（Graph Database）是基于图论实现的一种新型 NoSQL 数据库。它的数据存储结构和数据的查询方式都是以图论为基础的。图论中图的基本元素为节点和边，在图数据库中对应的就是节点和关系。

在图数据库中，数据与数据之间的关系通过节点和关系构成一个图结构并在此结构上实现数据库的所有特性，如对图数据对象进行创建、读取、更新、删除（Create、Read、Update、Delete，简称：CRUD）等操作的能力，还有处理事务的能力和高可用性等。

目前市面上较为流行的图数据库产品如图 1-2 所示。



图 1-2　较为流行的图数据库

1.2.2 图数据模型

图数据要具体存储到图数据库中，最终落实为具体的数据文件，自然就涉及特定的图数据模型，即如何存、采用什么实现方式来存图数据。常用的有三种：属性图（Property Graphs）、超图（Hypergraphs）和三元组（Triples）。下面分别讨论每种模型。

1.2.2.1 属性图

属性图模型直观更易于理解，能描述绝大部分图使用场景，也是当下最流行的图数据模型。Neo4j 采用的就是这种属性图模型。符合下列特征的图数据模型就称为属性图：

- 它包含节点和关系。
- 节点可以有属性（键值对）。
- 节点可以有一个或多个标签。
- 关系有名字和方向，并总是有一个开始节点和一个结束节点。
- 关系也可以有属性。

1.2.2.2 超图

超图是一种更为广义的图模型，在超图中，一个关系（称作超边）可以关联任意数量的节点，无论是开始节点端还是结束节点端，而属性图中一个关系只允许一个开始节点和一个结束节点，因此，超图更适用表示多对多关系。比如常见的房产拥有关系，如图 1-3 所示，在房产证上张三与李四共同拥有三套房，在超图中就只需一条超边（拥有）就能表示出来。

但现实中，仅仅一条超边来表示拥有关系，可能会隐藏很多细节，例如房产证中每套房张三、李四各自占有的比率，因此，如果用属性图来表示将更为丰富，只是将一条超边转化为 6 条属性图中的关系，如图 1-4 所示。

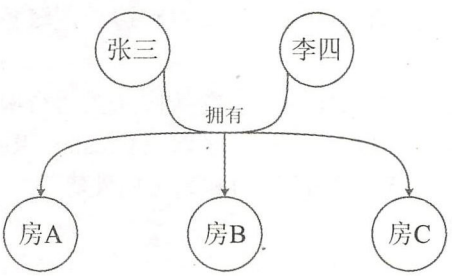


图 1-3 简单的房产拥有关系超图表示

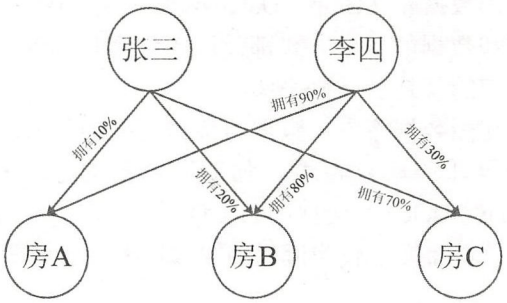


图 1-4 简单的房产拥有关系属性图表示

1.2.2.3 三元组

三元组思想来源于语义网（Semantic Web），虽然迄今为止，只有很少的网络资源是用语义网来表示的，但研究人员发现可以使用带语义的标签网来表示图数据。三元组是一个包含主谓宾的数据结构，例如张三和李四拥有三套房子等。显然，单个三元组的语义还是比较有限，需要借助资源描述框架（Resource Description Framework，RDF）来增强其知识推理及数据关联性。由于按三元组模型来实现的图数据库产品很少，在此不作进一步介绍，有兴趣的读者可以深入查阅语义网的相关资料。

1.2.3 图计算引擎

与关系数据库相同，图数据库的核心也是构建在一个引擎之上的，那就是图计算引擎。

图计算引擎是能够组织存储大型图数据集并且实现了全局图计算算法的一种数据库核心构件。

图 1-5 展示了一个图计算引擎的工作流程。它包括一个具有联机事务处理过程（On-Line Transaction Processing，OLTP）的数据库记录系统，图计算引擎用于响应用户终端或应用程序运行时发来的查询请求，周期性从记录系统中进行数据抽取、转换和加载，然后将数据从记录数据系统读入到图计算引擎并进行离线查询和分析，最后将查询、分析的结果返回给用户终端或应用程序。

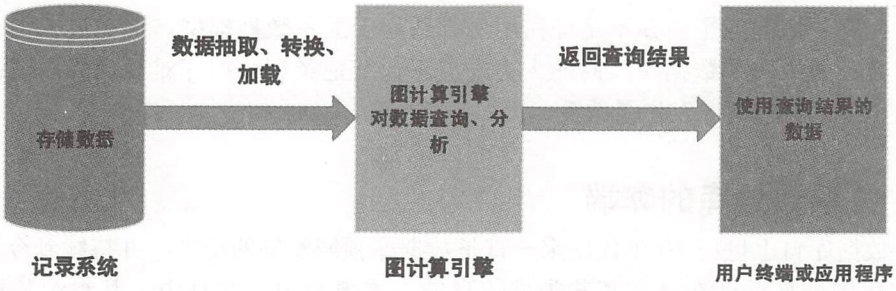


图 1-5　典型图计算引擎工作流程图

目前较为流行的图计算引擎有两种：单机图计算引擎和分布式图计算引擎。

单机图计算引擎的典型代表是 Cassovary，Cassovary 是一个用 Scala 编写的基于 Java 虚拟机的图计算引擎。在 Twitter 上 Cassovary 用来为其提供许多基于图的功能，如谁关注了谁、谁被谁关注。

分布式图计算引擎的典型代表是 Pegasus 和 Giraph。Pegasus 是一个运行在 Hadoop 云计算平台之上的分布式图计算引擎，最初它是为了 Google 的网页数据处理而设计出来的；Giraph 是 Apache 一个高可扩展性图计算项目，它最初起源于 Google 开发的 Pregel 图形处理项目。

1.2.4　图数据库的历史

在 20 世纪 60 年代中期，Navicational 数据库出现，它被认为是图数据库的雏形，这种数据库可以以树状结构的形式存储和组织数据。IBM 的 IMS 也是此类数据库的典型代表。

在 20 世纪 60 年代后期，数据库以网络模型的存储与组织形式逐渐衍生出图数据库的概念。1969 年，CODASYL 为 COBOL（一种数据处理领域最为广泛的程序设计语言）定义了网络数据库语言。

20 世纪 90 年代初，为了在网站应用中加速索引网页的性能，图数据库有了较大的改进。

在 2005 年前后，商业化的具有 ACID 特性——Atomicity（原子性）、Consistency（一致性）、Isolation（隔离性）、Durability（持久性）的图数据库出现了，如：Neo4j、Oracle Spatial、GraphDB 等。

在 2010 年，具有并行扩展功能的 ACID 图数据库出现，SAP HANA（High-Performance Analytic Appliance，一种高性能分析设备）将列式存储技术引入内存数据库。同期，可同时支持图形、关系、面向文档等多种存储形式的数据库也出现了，例如 OrientDB、ArangoDB 和 MarkLogic。

随着社交应用的普及，社交应用处理复杂网状关系的需求与图数据库的特性自然而然地相互结合了，各种图数据库成为较受欢迎的社交网络数据分析解决方案。

1.3 图数据库与关系数据库的对比

在数据库领域的应用中并不是任何用例都适用于关系数据模型，但在过去由于缺乏可行的替代方案和各大关系数据库厂商的大力发展下，其他类型数据库难以成为主流；但随着图数据库的产生，这种情况发生了改变。

1.3.1 关系数据库的弊端

关系数据库自上世纪 80 年代以来一直是数据库领域发展的动力，并持续到今天。它们将高度结构化的数据存储在具有某些类型信息的二维表格中，并且由于其组织数据的严格特性，开发人员和应用程序必须严格地按照关系数据库的相关约定来构建其应用程序中使用的数据。

在关系数据库中，通过外键约束来实现两表或多个表之间某些记录相互引用的关系。外键约束是关系数据库中实现表之间相互引用的必不可少的策略。关系数据库通过外键在主表中寻找匹配的主键记录来进行搜索、匹配计算操作，因为这种操作是“计算密集型”的，也是“内存密集型”的，并且操作次数将是表中记录的指数级别，所以它需要消耗大量的系统资源。如果您使用多对多关系，则必须要再添加一个中间表，它用来保存两个参与表的外键对应关系，这进一步增加了连接操作的成本。

1.3.2 图数据模型的优势

在图数据库中，关系是最重要的元素。通过关系我们能够将节点相互关联起来构建与我们的问题领域密切相关的复杂模型。

图数据库模型中的每个节点都直接包含一个关系列表，关系列表中存放此节点与其他节点的关系记录。这些关系记录按类型和方向组织起来，并且可以保存附加属性。无论何时运行类似关系数据库的连接（Join）操作时，图数据库都将使用此列表来直接访问连接的节点，无须进行记录的搜索、匹配计算操作。

将关系预先保存到关系列表中的这种能力使 Neo4j 能够提供比关系数据库高几个数量级的性能，特别是对于复杂连接的查询，Neo4j 能够实现毫秒级的响应。

使用图数据库来组织数据所得到的数据模型比使用传统关系或其他 NoSQL 数据库的数据模型更简单，同时更具有表现力。

图数据库支持非常灵活和细粒度的数据模型，可以用简单直观的方式对数据应用进行建模和管理，可以更方便地将数据单元小型化、规范化；同时还能实现丰富的关系连接，这样在对数据查询时可以用任何可想象到的方式进行查询操作。可见，与关系数据库相比，图数据库可支持更多类型的用例，如图 1-6 所示。

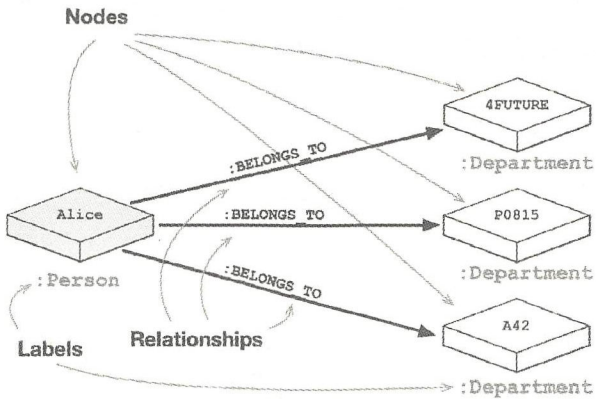


图 1-6　图数据库中关系的丰富表现

众所周知，数据库为确保数据的正确性，创造性地引入了事务概念，即 ACID 特性。像 Neo4j 这样的图数据库完全支持 ACID 事务，包括预写式日志（write-ahead logs）的恢复和异常终止后的恢复，所以永远不会丢失已经被加入数据库的数据。

下面是一个用关系数据模型和图数据模型建模的实例，通过两个 E-R 图（Entity Relationship Diagram，实体-联系图）可以看出使用图数据库更加简洁、明确地描述数据之间的关系。

如图 1-7 所示，要使用关系数据库创建一个项目部门组织的存储结构，必须为项目、部门、组织、人员单独创建表结构，各个表结构之间通过外键约束相互关联，对于多对多的复杂关系还必须创建“中间表”（如 Department_Members 表），通过 E-R 图可以看出，各个表结构中创建了大量的主键、外键列，并创建了中间表来维护复杂关系，这消耗了大量的系统资源。

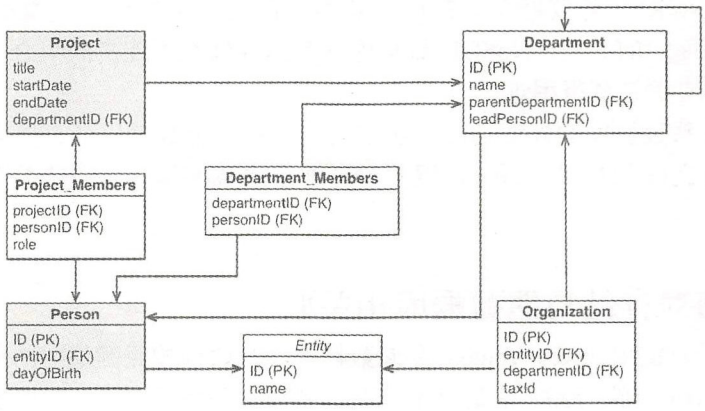


图 1-7　关系数据模型

如果采用图数据库创建一个项目部门组织的存储结构则容易得多，只需要为项目、部门、组织、人员创建节点，并且节点不需要主键、外键，也不需要中间表，只保留必要的属

性即可。各个节点直接通过关系指向来表达节点之间的复杂关系。使用图数据库可以更加简洁、明确地描述数据间的复杂关系，如图 1-8 所示。

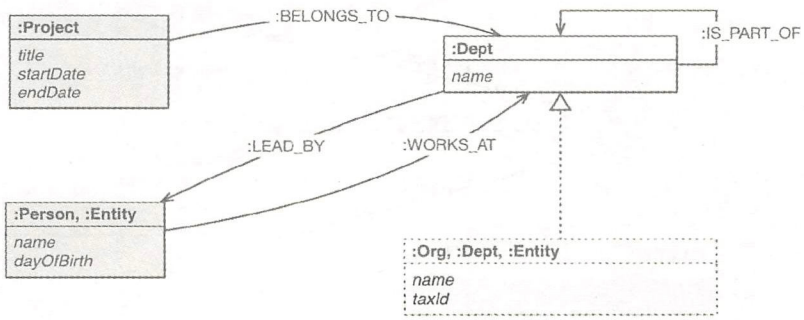


图 1-8　图数据模型

1.4 图数据库与其他 NoSQL 数据库的对比

为了克服关系数据库在大型企业级应用中的弊端，NoSQL 数据库逐渐流行起来。NoSQL 数据库专注于解决大规模数据集合、多重数据种类带来的挑战，尤其是大数据应用难题。同时 NoSQL 数据库的多样化也提供了多种不同的数据模型解决方案，每个解决方案都可以应用到不同的项目用例中。显然，Neo4j 就是 NoSQL 中可以提供图数据模型的典型代表。

1.4.1 其他 NoSQL 数据库的弊端

目前大部分 NoSQL 类数据库的数据存储形式都是基于集合的，数据按照集合被划分开，比如文档数据库中的文档。NoSQL 数据库的数据存储在不连贯的集合中，这使得数据之间的相互连接、建立关系变得困难。

要实现类似关系数据库的外键功能，通常人们会将某个数据集合直接嵌入到另一个数据集合中来实现两者之间的从属关系，但很明显这样创建数据集合之间的关系要付出很大的存储开销。

1.4.2 将键值对存储与图数据库相关联

如图 1-9 所示，键值（Key-Value）存储数据库，其数据按照键值对的形式进行组织、索引和存储，类似 Java 中的 map，它适合不涉及过多数据关系的应用。因为能有效减少读写磁盘的次数，所以其性能非常高。

如果将这些键值对相互关联，就可以得到一个图结构。通过这个图结构，可以表达数据之间的复杂关系，如图 1-10 所示。

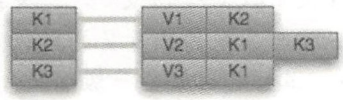


图 1-9　键值对存储结构

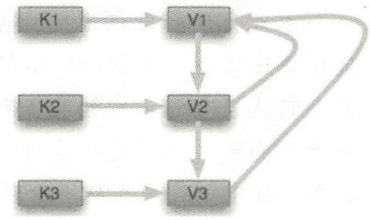


图 1-10　键值对存储结构与图结构相互关联

1.4.3　将文档存储与图数据库相关联

文档存储的层次化结构可容纳许多无模式的数据，可以轻松地将数据存储为树状结构。虽然树也是一种图形，但树只能表达从下到上的从属关系，如图 1-11 所示。

在树状存储结构中总会出现多次被嵌入的冗余数据，这增加了更新数据的困难，同时也难以确保数据的一致性。如果我们将冗余的数据去掉，然后用图结构将存在相互关系的数据相关联，数据冗余的问题就解决了，并且数据之间的关系变得更直观，如图 1-12 所示。

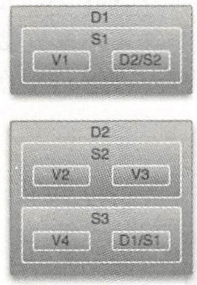


图 1-11　文档存储结构

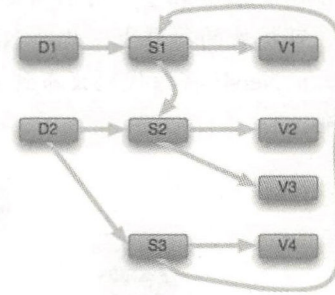


图 1-12　文档存储结构与图结构相互关联

1.5　Neo4j 概述

Neo4j 是由 Java 实现的开源 NoSQL 图数据库。自 2003 年开始研发，直到 2007 年正式发布第一版。Neo4j 的源代码托管在 GitHub 上，技术支持托管在 Stack Overflow 和 Neo4j Google 讨论组上。Neo4j 现如今已经被各种行业的数十万家公司和组织采用。Neo4j 的使用案例涵盖了网络管理、软件分析、科学研究、路由分析、组织和项目管理、决策制定、社交网络等诸多方面。

Neo4j 实现了专业数据库级别的图数据模型的存储。与普通的图处理或内存级数据库不同，Neo4j 提供了完整的数据库特性，包括 ACID 事务的支持、集群支持、备份与故障转移等，这使其适合于企业级生产环境下的各种应用。

另外，Neo4j 一些特殊功能使得 Neo4j 在用户、开发人员和 DBA 中非常受欢迎（如图 1-13 所示）。

- 一个本地化的图数据库：Neo4j 自底向上构建成一个图数据库。它的体系结构旨在优化快速管理、存储和遍历节点和关系。在 Neo4j 中，关系是数据库中最重要的元素，它代表节点之间的相互联系。众所周知，在关系数据库领域中，“关系”用于多个不同表之间的连接操作，这种操作的性能下降与关系的数量呈指数级别的，但在 Neo4j 中则是用于从一个节点指向另一个节点，其性能却是线性级别的。
- 界面友好：Neo4j 提供了一个查询与展示一体化的 Web 操作界面。对于图数据模型 Neo4j 使用 D3.js 做数据可视化，非常形象地展示了数据模型的节点和关系。
- 声明式图查询语言：Cypher 是一种声明式图数据库查询语言，它表现力丰富，查询效率高，其地位和作用与关系型数据库中的 SQL 语言（Structured Query Language，结构化查询语言）类似。Cypher 还有良好的扩展性，用户可以定制自己的查询方式（如自定义过程）。
- ACID 事务：Neo4j 通过 ACID 事务提供真正的数据安全，Neo4j 使用事务来保证数据在硬件故障或系统崩溃的情况下不会丢失。
- 高性能：Neo4j 使用多副本主从复制的方式构建高可靠性集群，支持大数据集合并且可以不断扩展其容量，可存储数百万亿个实体。也就是说，Neo4j 可部署在一个可容错、可扩展的集群上。此外 Neo4j 还提供热备份和性能监控功能。
- 代码开源：Neo4j 将源代码公布到 GitHub，任何人都可以去 Neo4j 的 GitHub 主页下载源代码。

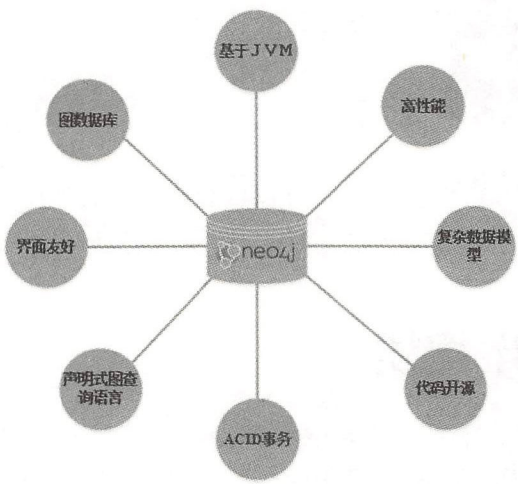


图 1-13　Neo4j 支持的特性

当然，传统的关系数据库经过长期的发展，其特性、性能已经被企业级应用认可，并且新兴的其他 NoSQL 数据库也越来越多地被使用在企业级项目的数据存储解决方案中；那么 Neo4j 这种图数据库与关系数据库、NoSQL 数据库对比，其优势是什么呢？ Neo4j 与其他数据库的对比情况如表 1-3 所示。

表 1-3　Neo4j 与其他数据库的对比

性能	关系数据库	其他 NoSQL 数据库	Neo4j
数据存储	关系数据库的数据存储在一个预定义好的、结构固定的二维表格中，数据之间的关系靠各个表格行列之间相互关联实现，查询效率低下	其他 NoSQL 不支持数据库级别的数据连接操作。性能和数据可信度随着连接的规模和复杂程度的增加而降低	Neo4j 采用具有自由邻接特性的图存储结构，能够提供更快的事务处理和数据关系处理功能
数据模型	关系数据库模型必须与建模者一起开发，并从逻辑模型转换为物理模型。由于数据类型和来源必须提前知道，所以后续的任何更改都会导致很大的结构变动，这对项目是很不利的	其他 NoSQL 数据模型不适合企业架构将其用来作为广泛的列和文档存储，不能在设计层面提供控制	Neo4j 提供灵活的数据模型，逻辑模型和物理模型之间不会相互耦合。可以随时、随意添加或更改数据和数据类型，从而大大缩短开发时间，实现项目真正的敏捷、迭代开发
数据查询性能	在关系数据库的多表联合查询中，数据处理性能严重受限于连接操作，如果单个表的行数过大，即便是很简单的连接查询也会耗费大量性能资源	其他 NoSQL 支持复杂数据关系的处理能力较弱，因此必须在应用程序级处理所有的数据关系	Neo4j 使用图存储结构，数据之间的关系附加在节点上，无论关系的数量或深度如何都能确保零延迟和实时性能
查询语言	SQL 查询语言，其复杂性会随连接数据查询所需的 JOIN 操作的数量而增加	各种 NoSQL 数据库的查询语言有所不同，但都没用于表达数据关系的机制	Cypher 是一种图查询语言，提供了描述关系查询的最有效的表达方式
事务处理	企业应用程序需要使用关系数据库的 ACID 事务支持，以确保数据的一致性和可靠性	其他 NoSQL 基本无法提供 ACID 事务，因为其数据关系的基本可用性和最终一致性是不可靠的	Neo4j 可以为企业应用程序提供 ACID 事务来保持全面一致和可靠的数据
数据库的扩展性	关系数据库通过主从服务器间的复制进行扩展，成本很高。如果数据表之间有多种复杂关系，这种扩展会降低集群的整体性能	其他 NoSQL 的扩展可以提升数据的写入性能但不能提升数据的读取性能，不能保证数据的可靠性	Neo4j 本质上适用于基于模式的查询。其扩展架构通过主从服务器间的复制来维护数据完整性。支持 IBM POWER8 和 CAPI Flash 系统进行大规模扩展
创建大规模数据中心	关系数据库可以通过服务器整合来创建大规模数据中心，但是其扩展架构是很昂贵的	使用 NoSQL 可以不断向其扩展架构中添加新硬件，但其不考虑能源成本、网络漏洞和其他风险	Neo4j 可以更高效地使用硬件，从而降低成本

1.6 Neo4j 的体系结构

本节将从数据库的底层设计揭开 Neo4j 数据库的神秘面纱，了解 Neo4j 为了实现图的存储所采用什么样的体系结构，揭秘为什么图数据库基于复杂联系的查询相比关系型数据库要更快。

Neo4j 最初的设计动机是为了更好地描述实体之间的联系。现实生活中，每个实体都与周围的其他实体有着千丝万缕的关系，这些关系里存在着大量的潜在信息。但是传统的关系型数据库更加注重刻画实体内部的属性，实体与实体之间的关系主要通过外键来实现。因

此，在查询一个实体的关系时需要 join 操作，特别是深层次的关系查询需要大量的 join 操作，而 join 操作通常又非常耗时。随着现实世界中关系数据的急剧增加，导致关系型数据库已经逐渐地难以承载查询海量数据深层次关系需要大量数据库表操作带来的运算复杂性，Neo4j 在这样的情况下应运而生。

1.6.1 免索引邻接

Neo4j 有一个重要的特点，就是用来保证关系查询的速度，即免索引邻接属性，数据库中的每个节点都会维护与它相邻节点的引用。因此每个节点都相当于与它相邻节点的微索引，这比使用全局索引的代价要小得多。这就意味着查询时间和图的整体规模无关，只与它附近节点的数量成正比。在关系型数据库中使用全局索引连接各个节点，这些索引对每个遍历都会增加一个中间层，因此会导致非常大的计算成本。而免索引邻接为图数据库提供了快速、高效的图遍历能力。

对比图 1-14 和图 1-15，可以明显地看出关系型数据库与 Neo4j 在查找关系时的区别。

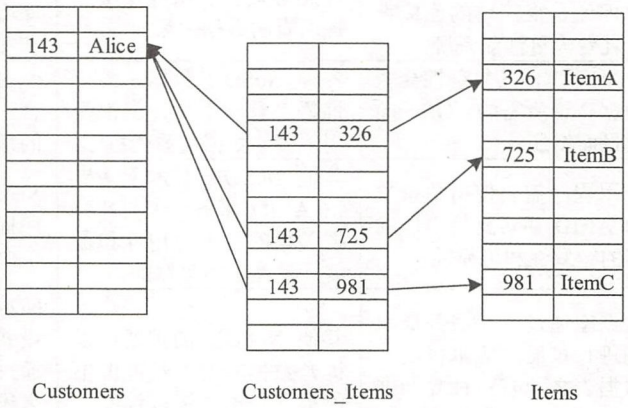


图 1-14 RDBMS 中关系查询示意图

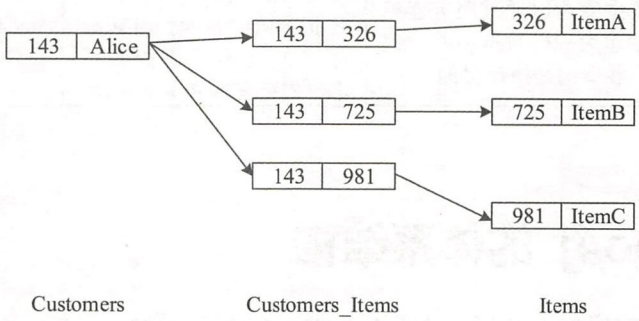


图 1-15 Neo4j 中关系查询示意图

图 1-14 展示了在 RDBMS（关系型数据库）中的查询方式。要查找 Alice 所购买的东西，首先要执行关系表的索引查询，时间成本为 $O(\log(n))$，n 为索引表的长度。这对于偶尔的浅层次查询是可以接受的，但是当查询的层次变深或者是执行反向查询时代价将会变得不

可接受了。如果相较于查询 Alice 购买的东西，要查询某件商品被哪些人购买了（推荐引擎中常用的一个场景），将不得不使用暴力方法来遍历整个索引，时间复杂度则将增长到 $O(n)$。除非我们再建立一个从商品到用户的索引表，但是该方法将会占用许多额外空间，并且使索引变得难以维护。

如果我们再考虑一个更复杂的场景，Alice 购买过的商品被哪些人购买过（推荐引擎中查找有共同爱好的人），找到 Alice 购买过的商品的时间成本为 $O(\log(n))$，找到每个商品被哪些人购买的时间成本为 $O(n)$，假如 Alice 购买过 m（m 远小于 n）个商品，那么总的时间复杂度即为 $O(mn\log(n))$。即使再建立一个方向索引表，时间复杂度也为 $O(mn)$。

图 1-15 展示了在同样的场景下 Neo4j 的查询方式。使用免索引近邻机制，每个节点都有直接或间接指向其相邻节点的指针。要查找 Alice 买过的东西，只需要在 Alice 的关系链表中遍历，每次的遍历成本仅为 $O(1)$。要查找一个商品被哪些人购买了，只要跟随指向该商品的关系来源即可，每次的遍历成本也是 $O(1)$。更复杂的，要查找 Alice 购买过的东西被哪些人购买过，时间复杂度也仅为 $O(m)$，其中 m 远小于 n。这相较于 RDBMS 的时间复杂度还是占有绝对的优势。

免索引邻接针对 RDBMS 中的关系查询的两个缺点做了改进：

（1）免索引邻接使用遍历物理关系的方法查找，比起全局索引来说代价要小得多。查询一个索引一般的时间复杂度为 $O(\log(n))$，而遍历物理关系的时间复杂度仅为 $O(1)$，至少对于 Neo4j 的存储结构来说是如此。

（2）当索引建立之后在试图反向遍历时，建立的索引就起不到作用了。我们有两个选择：对每个反向遍历的场景创建反向查找索引，或者使用原索引进行暴力搜索，而暴力搜索的时间复杂度为 $O(n)$。这种代价相对于很多需要实时操作的场景来说是不可接受的。

利用免索引邻接机制，在图数据库上进行关系查询效率非常高，这种高效是建立在图数据库注重关系的架构设计之上的。

1.6.2　Neo4j 底层存储结构

免索引邻接是图数据库实现高效遍历的关键，那么免索引邻接的实现机制就是 Neo4j 底层存储结构设计的关键。能够支持高效的、本地化的图存储以及支持任意图算法的快速遍历，是使用图数据库的重要原因。本节将深入地探索 Neo4j 的底层存储结构。

从宏观角度来说，Neo4j 中仅仅只有两种数据类型：

（1）节点（Node）：节点类似于 E-R 图中的实体（Entity）。每一个实体可以有零个或多个属性（Property），这些属性以 key-value 对的形式存在。属性没有特殊的类别要求，同时每个节点还具有相应的标签（Label），用来区分不同类型的节点。

（2）关系（Relationship）：关系也类似于 E-R 图中的关系（Relationship）。一个关系有起始节点和终止节点。另外，与节点一样，关系也能够有自己的属性和标签。

节点和关系在 Neo4j 中如图 1-16 所示。

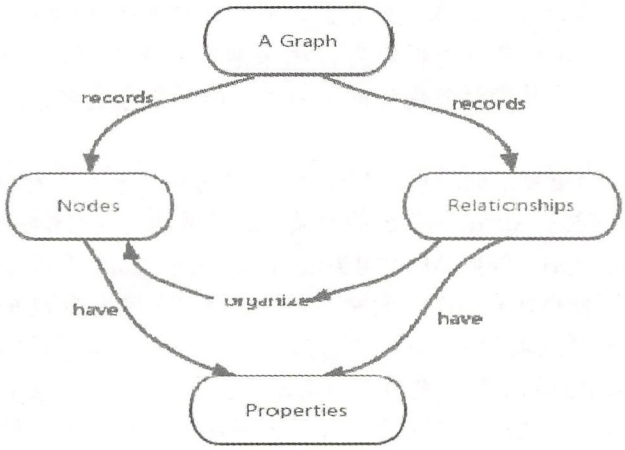


图 1-16　Neo4j 基本存储结构示意图

节点和关系分别采用固定长度存储，图 1-17 为 Neo4j 节点和关系存储文件的物理结构。

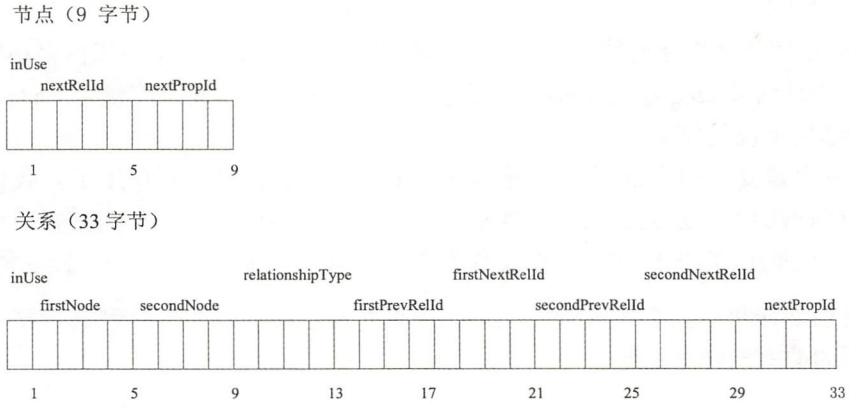


图 1-17　Neo4j 节点和关系存储文件的物理结构

节点存储文件用来存储节点的记录，文件名为 neostore.nodestore.db。节点记录的长度为固定大小，如图中所示，每个节点记录的长度为 9 字节。格式为：Node:inUse+nextRelId+nextPropId。

- inUse：1 表示该节点被正常使用，0 表示该节点被删除。
- nextRelId：该节点的下一个关系 ID。
- nextPropId：该节点的下一个属性 ID。

我们可以将 neostore.nodestore.db 中存储的记录看作是如下方式：

```
Node[0,used=true,rel=9,prop=-1]
Node[1,used=true,rel=1,prop=0]
Node[2,used=true,rel=2,prop=2]
Node[3,used=true,rel=2,prop=4]
Node[4,used=true,rel=4,prop=6]
```

```
Node[5,used=true,rel=5,prop=8]
Node[6,used=true,rel=5,prop=10]
Node[7,used=true,rel=7,prop=12]
Node[8,used=true,rel=8,prop=14]
Node[9,used=true,rel=8,prop=16]
Node[10,used=true,rel=10,prop=18]
Node[11,used=true,rel=11,prop=20]
```

Node[12,used=true,rel=11,prop=22]采用固定字节长度的记录可以快速地查询到存储文件中的节点。如果有一个 ID 为 100 的节点，我们知道该记录在存储文件中的第 900 字节。基于这种查询方式，数据库可以直接计算一个记录的位置，其成本仅为 $O(1)$，而不是执行成本为 $O(\log(n))$的搜索。

关系存储文件用来存储关系的记录，文件名为 neostore.relationshipstore.db。像节点的存储一样，关系存储区的记录大小也是固定的。格式为：Relationship:inUse+firstNode+secondNode+relType+firstPrevRelId+firstNextRelId+secondPrevRelId+secondNextRelId+nextPropId。

- inUse，nextPropId：作用同上。
- firstNode：当前关系的起始节点。
- secondNode：当前关系的终止节点。
- relType：关系的类型。
- firstPrevRelId & firstNextRelId：起始节点的前一个和后一个关系的 ID。
- secondPrevRelId & secondNextRelId：终止节点的前一个和后一个关系 ID。

同样的，“neostore.relationshipstore.db”中存储的记录也可以看作是如下的方式：

```
Relationship[0,used=true,source=1,target=0,type=0,sPrev=1,sNext=-
1,tPrev=3,tNext=-1,prop=1]
Relationship[1,used=true,source=2,target=1,type=1,sPrev=2,sNext=-1,tPrev=-
1,tNext=0,prop=3]
Relationship[2,used=true,source=3,target=2,type=2,sPrev=-1,sNext=-1,tPrev=-
1,tNext=1,prop=5]
Relationship[3,used=true,source=4,target=0,type=0,sPrev=4,sNext=-
1,tPrev=6,tNext=0,prop=7]
Relationship[4,used=true,source=5,target=4,type=1,sPrev=5,sNext=-1,tPrev=-
1,tNext=3,prop=9]
Relationship[5,used=true,source=6,target=5,type=2,sPrev=-1,sNext=-1,tPrev=-
1,tNext=4,prop=11]
Relationship[6,used=true,source=7,target=0,type=0,sPrev=7,sNext=-
1,tPrev=9,tNext=3,prop=13]
Relationship[7,used=true,source=8,target=7,type=1,sPrev=8,sNext=-1,tPrev=-
1,tNext=6,prop=15]
Relationship[8,used=true,source=9,target=8,type=2,sPrev=-1,sNext=-1,tPrev=-
1,tNext=7,prop=17]
Relationship[9,used=true,source=10,target=0,type=0,sPrev=10,sNext=-1,tPrev=-
1,tNext=6,prop=19]
Relationship[10,used=true,source=11,target=10,type=1,sPrev=11,sNext=-
```

```
1,tPrev=-1,tNext=9,prop=21]
Relationship[11,used=true,source=12,target=11,type=2,sPrev=-1,sNext=-
1,tPrev=-1,tNext=10,prop=23]
```

Neo4j 中有一个 .id 文件用来保持对未使用记录的跟踪，用来回收未使用的记录占用的空间。节点和关系的存储文件只关心图的基本存储结构而不是属性数据。这两种记录都使用固定大小的记录，以便存储文件内的任何记录都可以根据 ID 快速地计算出来。这些都是强调 Neo4j 高性能遍历的关键设计决策。节点记录和关系记录都是相当轻量级的：只由几个指向联系和属性列表的指针构成。

图 1-18 是 Neo4j 中的其他常见的基本存储类型，需要注意的是属性的存储，属性记录的物理存储放置在 neostore.propertystore.db 文件中。与节点和关系的存储记录一样，属性的存储记录也是固定长度。每个属性记录包含 4 个属性块和属性链中下一个属性的 ID。属性链是单向链表，而关系链是双向链表。一个属性记录中可以包含任何 Java 虚拟机（JVM）支持的基本数据类型、字符串、基于基本类型的数组以及属性索引文件（neostore.propertystore.db.index）。属性索引文件主要用于存储属性的名称，属性索引的值部分存储的是指向动态内存的记录或者内联值，短字符串和短数组会直接内联在属性存储记录中。当长度超过属性记录中的 propBlock 长度限制之后，会单独存储在其他的动态存储文件中。

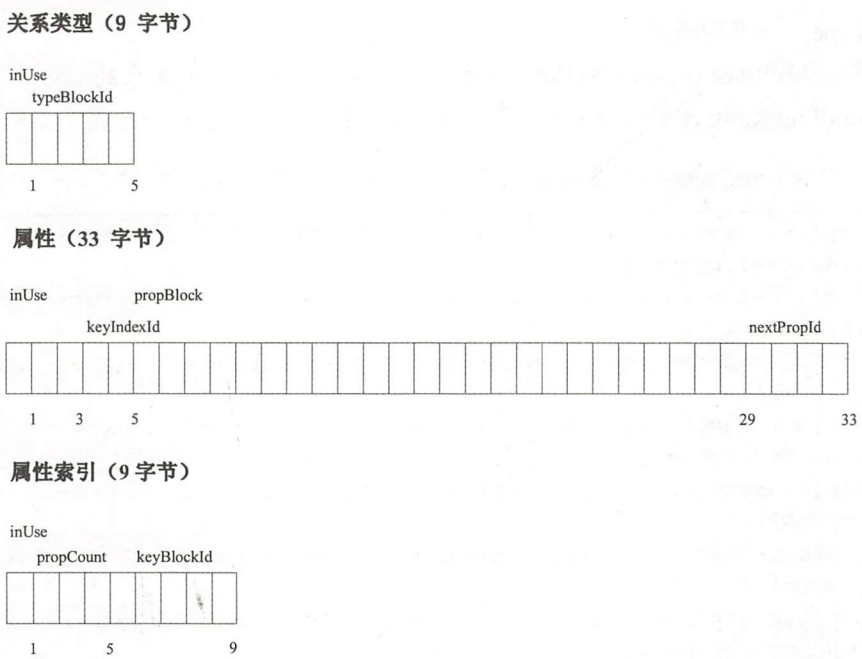


图 1-18　Neo4j 其他常见的物理结构

Neo4j 中有两种动态存储：动态字符串存储（neostore.propertystore.db.strings）和动态数组存储（neostore.propertystore.db.arrays）。动态存储记录是可以扩展的，如果一个属性长到一条动态存储记录仍然无法完全容纳时，可以申请多个动态存储记录逻辑上进行连接。

1.6.3　Neo4j 的遍历方式

我们可以看到磁盘上各种存储文件存在交互。每个节点记录都包含一个指向该节点的第一个属性的指针和联系链中第一个联系的指针。要读取一个节点的属性，从指向第一个属性的指针开始，遍历整个单向链表结构。要找到一个节点的关系，从指向的第一个关系开始，遍历整个双向链表，直到找到了感兴趣的关系。一旦找到了感兴趣关系的记录，我们就可以与使用和查找节点属性一样的方法查找关系的属性（如果该关系存在属性）。我们也可以很方便地获取起始节点和结束节点的 ID，利用节点 ID 乘以节点记录的大小（9 字节）就可以立即算出每个节点在节点存储文件中的具体位置，所需的复杂度仅为 $O(1)$。

图 1-19 就是节点、关系、属性在 Neo4j 中的物理表现方式。图中直接看起来可能不是那么直观，可以想象关系是属于两个节点的，但是一个关系也不应该在记录中出现两次，这样会造成内存的浪费。因此两个双向链表之间会有指针，一个是起始节点可见的列表关系，另一个是结束节点可见的列表关系。每一个列表都是双向链表，因此我们可以在任何一个方向上进行快速遍历和高效地插入和删除。

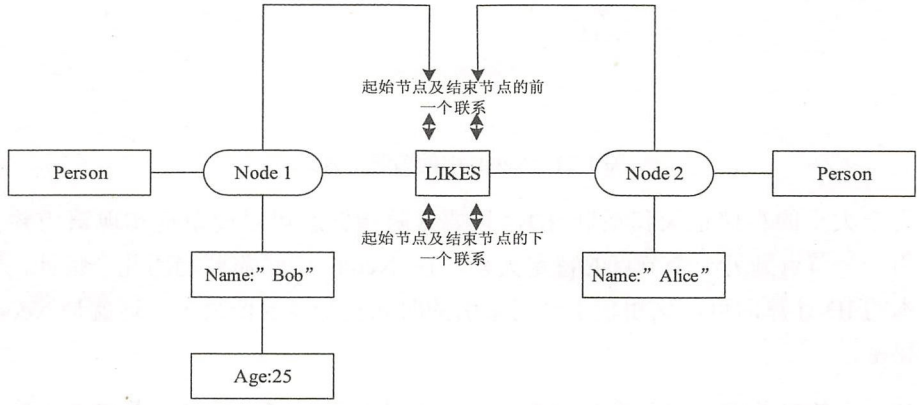


图 1-19　在 Neo4j 中的物理存储方式

下面通过一个例子来讲解 Neo4j 遍历关系和节点的详细过程。假如在 Neo4j 中存储了 A、B、C、D、E 5 个节点和 R1、R2、R3、R4、R5、R6、R7 7 个关系，它们之间的关系如图 1-20 所示。

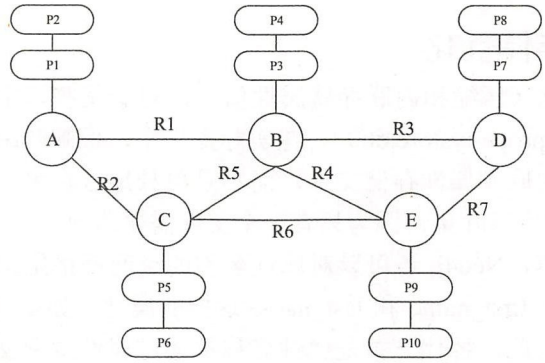


图 1-20　Neo4j 中的一个关系结构示意图

假如要遍历图中节点 B 的所有关系，只需要向 NODEB-NEXT 方向遍历，直到指向 NULL 为止，可以从图 1-21 中看出节点 B 的所有关系为 R1、R3、R4、R5。

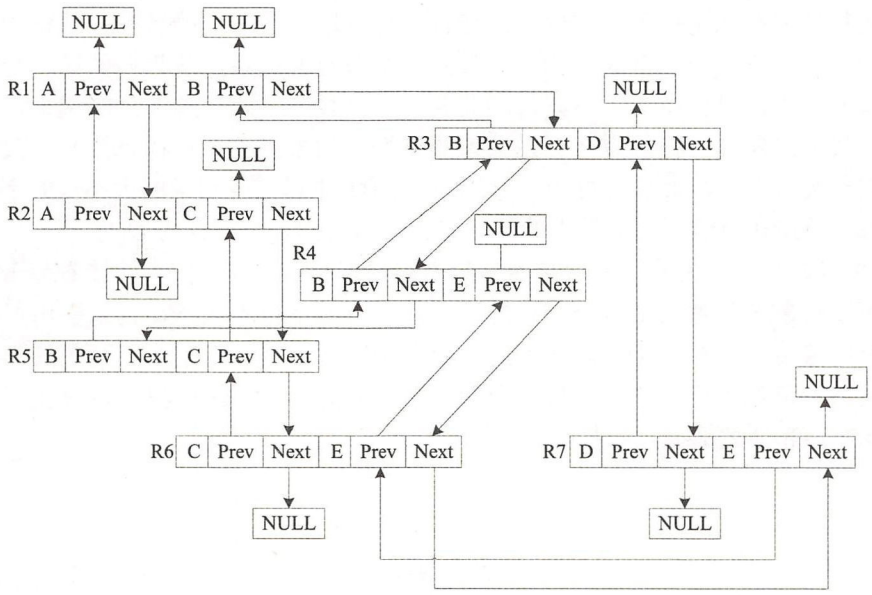


图 1-21　Neo4j 中的图遍历方式

通过固定大小的存储记录和指针 ID，只要跟随指针就可以简单地实现遍历并且高速执行。要遍历一个节点到另一个节点的特定关系，在 Neo4j 中只需要遍历几个指针，然后执行一些低成本的 ID 计算即可，这相较于全局索引的时间复杂度要低很多，这就是 Neo4j 实现高效遍历的秘密。

- 从一个给定节点定位关系链中第一个关系的位置，可以通过计算它在关系存储的偏移量来获得。跟获得节点存储位置的方法一样，使用关系 ID 乘以关系记录的固定大小即可找到关系在存储文件中的正确位置。
- 在关系记录中，搜索第二个字段可以找到第二个节点的 ID。用节点记录大小乘以节点 ID 可以得到节点在存储中的正确位置。

1.6.4　Neo4j 的存储优化

Neo4j 支持存储优化（压缩和内联存储属性值），对于某些短字符的属性可以直接存储在属性文件中（neostore.propertystore.db）。在实际操作中，像邮政编码、电话号码这样的短字符串属性就可以直接内联到属性存储文件，而不是单独地放在另一个动态存储区，这样将大幅减少 I/O 操作并且增大吞吐量，因为只有一个文件需要访问。

除了可以内联属性值，Neo4j 还可以对属性名称的空间严格维护，例如在社交网络中，有可能会有多个节点存在 first_name 和 last_name 这样的属性。如果将每个属性都逐字写入到磁盘上就会造成浪费。因此，替代方案是属性名称都通过属性索引文件从属性存储中间接引用。属性索引允许所有具有相同名称的属性共享单个记录，因而 Neo4j 可以节省相当大的空

间和 I/O 开销。

即使现在的磁盘访问速度已经很快了，但是 CPU 访问磁盘仍然比 CPU 直接访问高速缓存要慢得多。因此，Neo4j 也采用了缓存策略，保证那些经常访问的数据可以快速地被多次重复访问。Neo4j 高速缓存的页面置换算法是基于最不经常使用的页置换（LFU，Least Frequently Used）缓存策略，根据页的常用程度进行微调。也就是说即使有些页面近期没有使用过，但是因为以前的使用频率很高，那么在短期之内它也是不会被淘汰的。该策略保证了缓存资源在统计学上的最优配置。



第 2 章

◂Neo4j 基础入门▸

本章包括以下内容：

- Neo4j 安装部署
- Neo4j Web 管理平台
- Neo4j 图数据中基本元素与概念
- 官方入门实例简介
- 批量导入工具

本章主要内容是引导读者如何快速了解和使用 Neo4j，包括 Neo4j 的安装部署、管理平台的使用、入门实例简介等。为满足初学者的好奇心，便于尝试把已有关系数据库中的数据导入到 Neo4j，在本章最后介绍如何使用批量导入工具。

2.1 Neo4j 的安装部署

Neo4j 数据库支持安装部署的操作系统非常广泛，如 Windows、Mac、CentOS、Ubuntu 等系统均可安装。本节将按照操作步骤逐步介绍安装 Neo4j 的过程，涉及安装部署的具体系统需求、各个端口的作用、密码设置等细节请参见“5.1 部署”节，本节依照步骤操作即可将 Neo4j 安装在自己的电脑上。



由于 Neo4j 是基于 Java 虚拟机（Java Virtual Machine，JVM）的产品，所以在安装前，必须保证已经安装了 Java 虚拟机，对于 Java 虚拟机的安装，请查阅其他相关的网上教程。

2.1.1 Neo4j 安装包的下载

首先，可访问 Neo4j 官方网站：https://neo4j.com/，找到下载链接，如图 2-1 所示，在官方网站首页有个 Download Now 按钮，单击后进入版本选择页面。



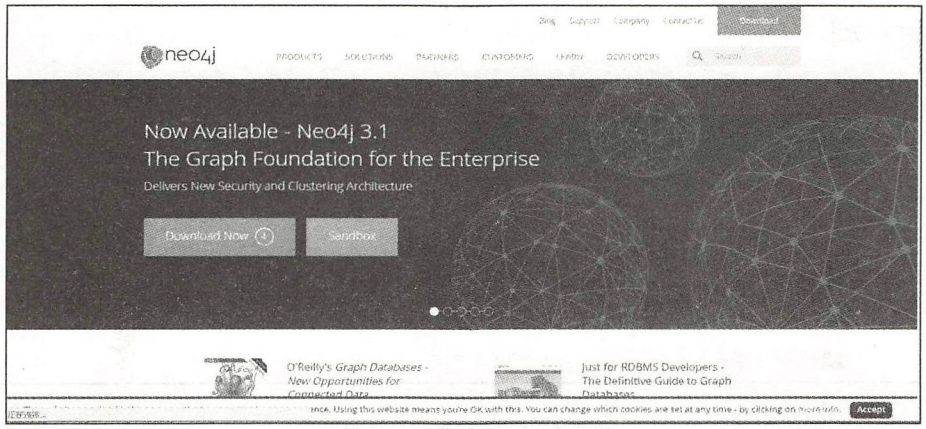


图 2-1　Neo4j 官方网站首页

在版本选择页面，可选择企业版或社区版，作为初学者选择社区版就足够了，且社区版免费，选择 Download Community Edition，如图 2-2 所示。

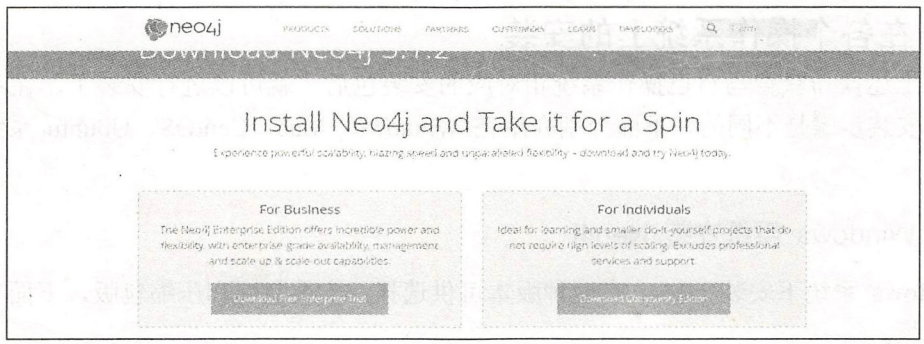


图 2-2　Neo4j 官方网站版本选择页

之后进入下载页面，在本页面可以选择相应操作系统的 Neo4j 版本。默认情况下此页面将推荐与操作系统最匹配的版本，如图 2-3 所示。若使用 Windows 系统，页面则推荐下载 Neo4j 的 Windows 安装版本。

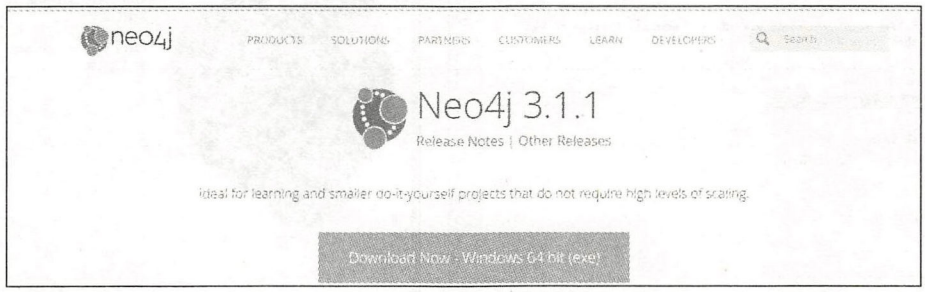
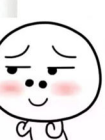


图 2-3　Neo4j 官方网站下载页

另外，可以单击 Other OS Downloads 选项，来选择 Linux、Mac OS，或者 Windows 系统 Zip 压缩包版本，如图 2-4 所示。



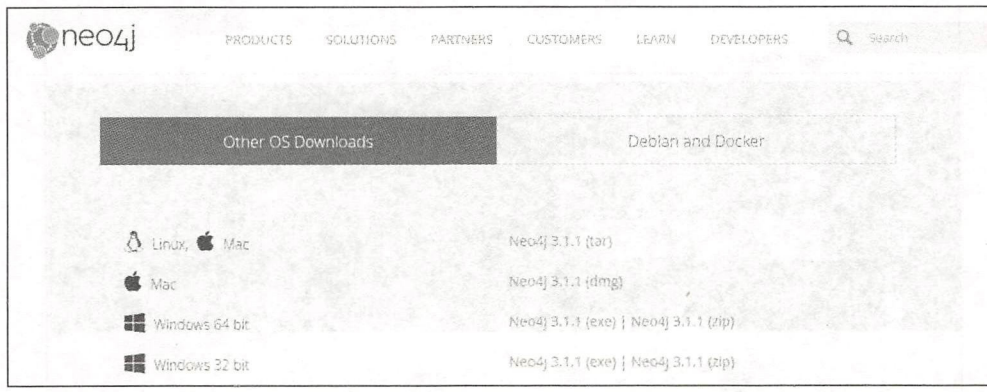


图 2-4　操作系统对应下载页

针对 Linux 系统，可以单击下载 tar 格式的压缩包，或者在 Linux 系统中使用命令下载（后续“2.1.2 在各个操作系统上的安装节将会介绍）。

针对 Mac 系统，就选择 dmg 安装包版本或者 tar 压缩包版本。

2.1.2　在各个操作系统上的安装

通过上述操作获得与自己操作系统相对应的安装包后，就可以进行安装了。在不同的操作系统中安装步骤是不同的，下面分别介绍在 Windows、Mac、CentOS、Ubuntu 系统下的安装步骤。

2.1.2.1　Windows 系统安装 Neo4j

Windows 系统下安装 Neo4j 有两种版本可供选择：安装包版和压缩包版。下面我们先介绍安装包版。

按照普通安装包的安装方式，双击打开安装包，按照引导选择安装路径，然后单击 Next 按钮，如图 2-5 和图 2-6 所示。

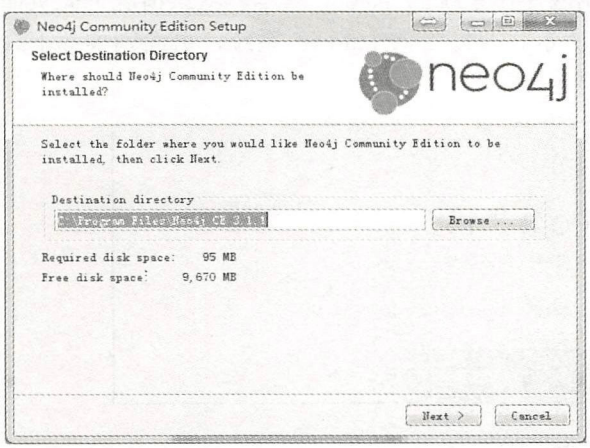


图 2-5　Neo4j 安装路径选择界面



图 2-6　Neo4j 安装欢迎界面

在此接受安装协议，单击 Next 按钮，在安装设置页保持默认选项，单击 Next 按钮安装

仅供非商业用途或交流学习使用

程序就开始安装了，安装完毕后可以选择打开 Neo4j 启动器，如图 2-7~图 2-10 所示。

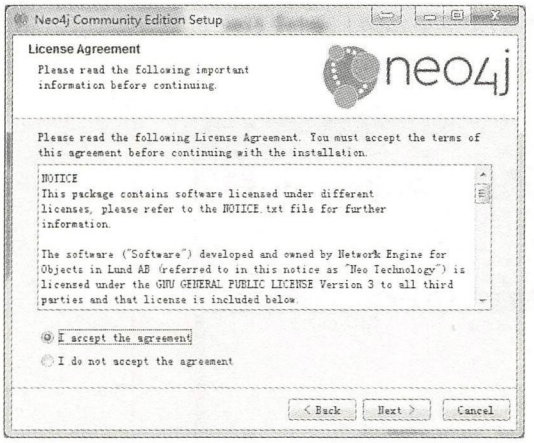


图 2-7　Neo4j 安装确认协议页

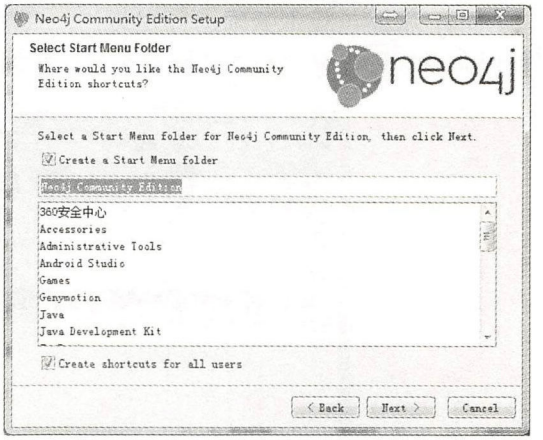


图 2-8　Neo4j 安装开始菜单设置

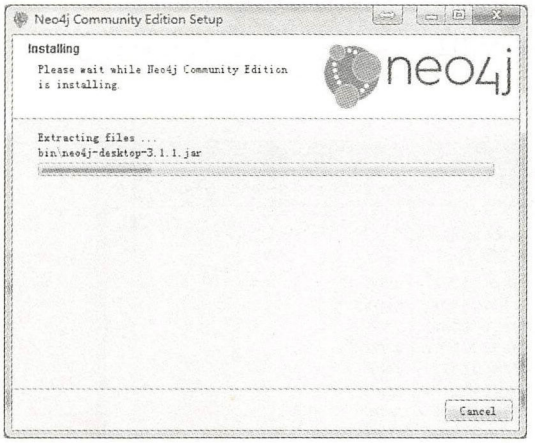


图 2-9　Neo4j 安装进度界面

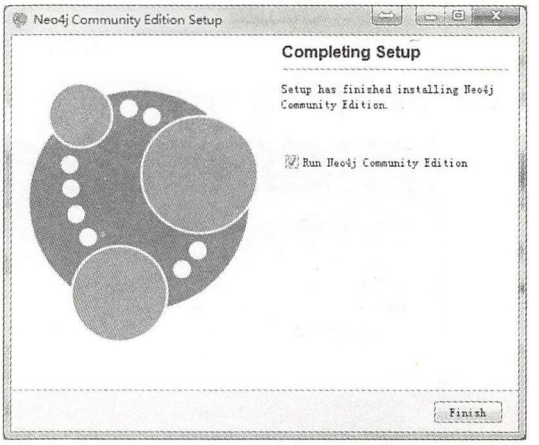


图 2-10　Neo4j 安装完成界面

至此，Windows 系统下的 Neo4j 安装包版已安装完毕。您可以单击 Neo4j 启动器的 Start 按钮来启动数据库，如图 2-11 所示。

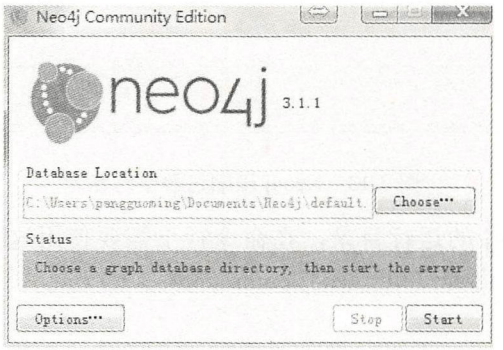


图 2-11　Neo4j 启动、关闭控制界面

仅供非商业用途或交流学习使用

2.1.2.2 Windows 系统下压缩包版安装 Neo4j

Neo4j 压缩包版本无须运行安装包，只需要将压缩文件解压到任意文件夹下即可，但需要使用命令行进行一些相关的配置来启动数据库，如图 2-12 所示。

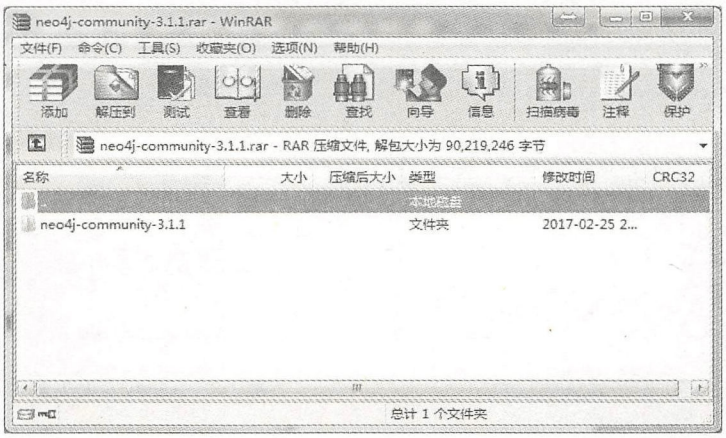


图 2-12 Neo4j 压缩包

解压后程序目录如图 2-13 所示。

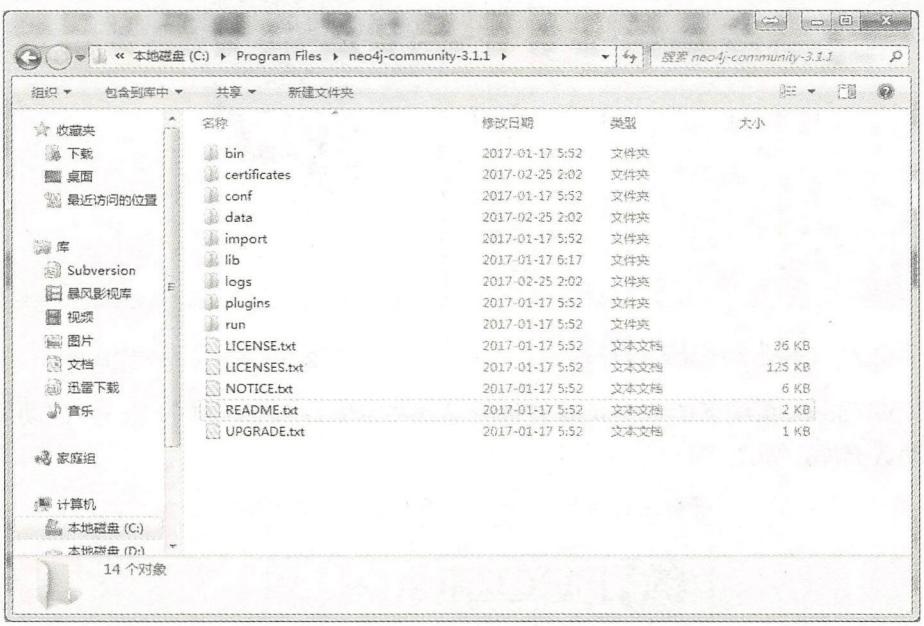


图 2-13 Neo4j 压缩包解压内容

在 bin 目录下是 Neo4j 的运行目录，在命令行下需要使用这个目录的文件地址，如图 2-14 所示文件地址是 C:\Program Files\neo4j-community-3.1.1\bin。



仅供非商业用途或交流学习使用

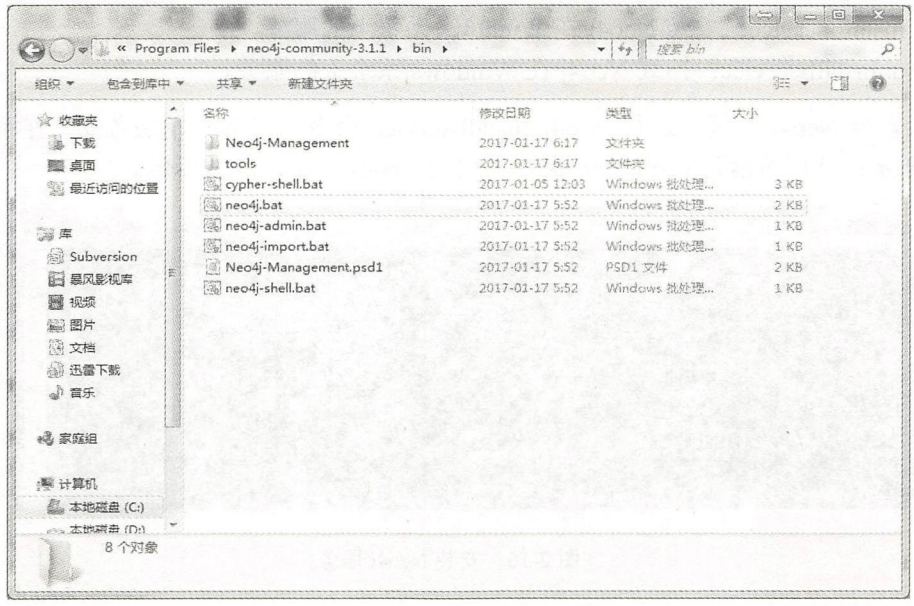


图 2-14　Neo4j 压缩包版执行路径

接下来在命令行下进行配置。首先需要使用管理员身份启动命令行工具，具体操作是在“开始”菜单找到“附件”下的“命令提示符”，右击后选择“以管理员身份运行”。

运行后输入 cd C:\Program Files\neo4j-community-3.1.1\bin 命令导航到 Neo4j 的运行目录，如图 2-15 所示。

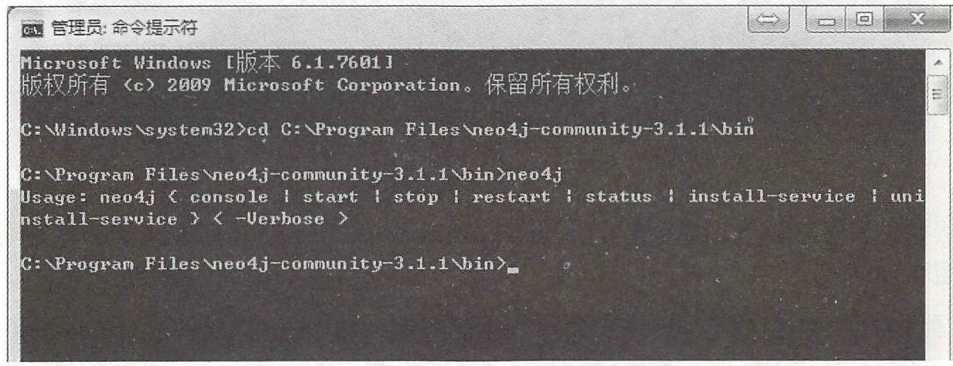


图 2-15　通过控制台导航到 Neo4j 执行路径

在 Neo4j 运行目录下输入命令：neo4j，系统会返回关于 neo4j 运行命令的相关指令，按照“neo4j <指令名>”的格式就可以运行相关操作了。

- console：打开 Neo4j 的控制台。
- start：启动 Neo4j。
- stop：关闭 Neo4j。
- restart：重启 Neo4j。
- status：查看 Neo4j 运行状态。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

- install-service：安装 Neo4j 在 Windows 系统上的服务。
- uninstall-service：卸载 Neo4j 在 Windows 系统上的服务。

首次安装 Neo4j 需要先运行 neo4j install-service 命令，将 Neo4j 服务安装在系统上，如图 2-16 所示；然后再运行 neo4j start 命令，启动 Neo4j，如图 2-17 所示。



图 2-16　安装 Neo4j 服务



图 2-17　启动 Neo4j

当然如果想要卸载 Neo4j，就运行 neo4j uninstall-service 命令，这样 Neo4j 服务就卸载了，如图 2-18 所示。

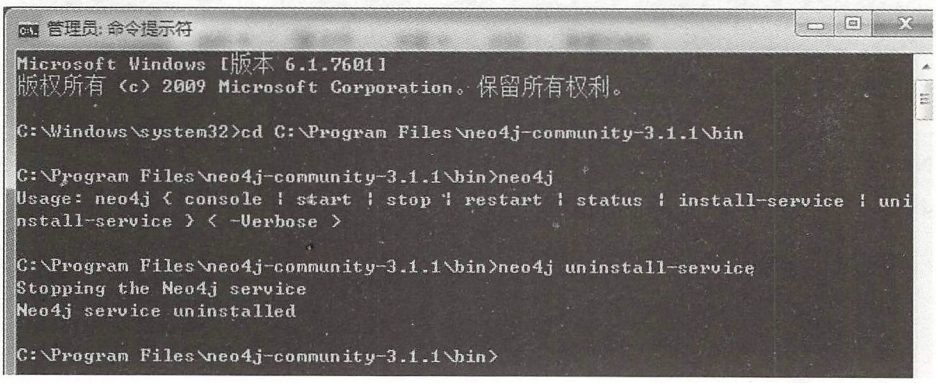


图 2-18　卸载 Neo4j 服务

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

2.1.2.3　Mac 系统安装 Neo4j

Mac 系统的 dmg 安装相对简单，只需要将 dmg 安装包拖入到名为 Applications 的应用程序文件夹下就可以了，如图 2-19 所示。

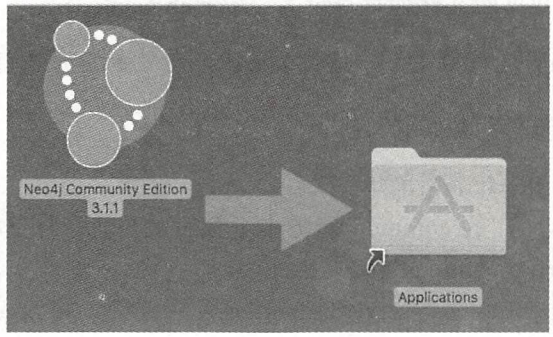


图 2-19　Mac 系统安装 Neo4j

2.1.2.4　CentOS 系统安装 Neo4j

对于 CentOS 系统安装 Neo4j，可以全部通过命令来完成，这样方便以后在很多实际生产环境下能部署到没有安装图形界面的 CentOS 系统上。

在 CentOS 命令行下依次输入如下命令：

```
cd /tmp
wget http://debian.neo4j.org/neotechnology.gpg.key
sudo rpm --import neotechnology.gpg.key
```

其中 cd /tmp 为导航到系统 tmp 目录下。

然后使用 wget http://debian.neo4j.org/neotechnology.gpg.key 命令将安装配置文件 neotechnology.gpg.key 下载到当前目录；

再使用 sudo rpm --import neotechnology.gpg.key 命令将安装配置文件导入到系统中。

接下来，需要编辑 neo4j.repo 文件，用文本编辑器创建一个 neo4j.repo 文件并在此文件内填入下列内容：

```
[neo4j]
name=Neo4j Yum Repo
baseurl=http://yum.neo4j.org
enabled=1
gpgcheck=1
```

将已经创建好的 neo4j.repo 文件添加到系统 yum repo 安装环境中的命令如下：

```
nano neo4j.repo
sudo cp neo4j.repo /etc/yum.repos.d/
```

最后，就可以使用 yum 命令安装 neo4j。

```
sudo yum install neo4j
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

至此在 CentOS 系统下 Neo4j 已安装完毕。下面是安装后 Neo4j 的文件路径：

（1）Neo4j 安装目录为：/usr/share/neo4j。

（2）Neo4j 的属性文件所在目录为：/etc/neo4j。

（3）Neo4j 默认的数据库文件保存目录为：/var/lib/neo4j。

在/usr/share/neo4j/bin 运行目录下，运行 neo4j start 命令就可以启动 neo4j 数据库了。

2.1.2.5 Ubuntu 系统安装 Neo4j

Ubuntu 系统下安装 Neo4j 非常简单。

（1）第一步：将 Neo4j 的 community 版本下载后，在命令行下将 tar 文件进行解压，运行的命令是：tar -zvxf neo4j-community-2.0.1.tar.gz。

（2）第二步：修改 conf/neo4j-server.properties 配置文件，将 org.neo4j.server.webserver.address=0.0.0.0 注释字符去掉。

（3）最后：进入 bin 目录，运行 neo4j start 命令启动 neo4j 数据库即可。

2.1.3 Neo4j 的启动

2.1.3.1 启动方式

在 Windows 和 Mac 系统下，直接运行安装完后的 Neo4j 启动器，就可以看到如下操作界面，单击 Start 按钮就可以启动，如图 2-20 所示。

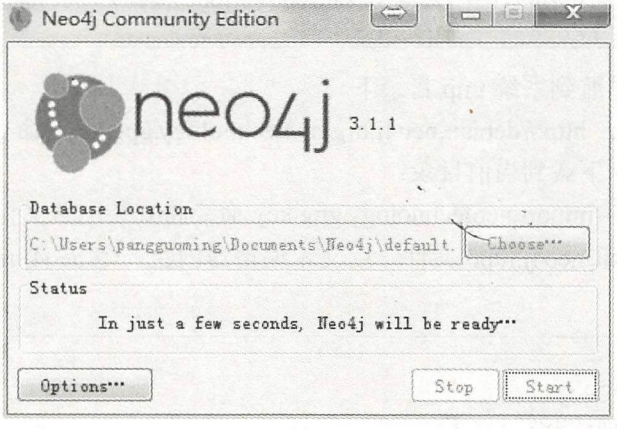


图 2-20 Neo4j 在 Windows 系统上的启动、关闭界面（Mac 系统上的界面相同）

在 CentOS 和 Ubuntu 下，需要在 Neo4j 的运行目录下，再运行 neo4j start 命令。

2.1.3.2 启动后的操作界面

启动后，可用浏览器打开网址：http://localhost:7474/，就可以看到 Neo4j 的操作界面。

在首次打开 Neo4j 操作界面时，需要修改默认的用户名和密码。默认情况下，用户名和密码都为 neo4j，输入默认的用户名、密码后单击 connect 按钮，就进入密码修改页面，然后

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

输入自己想要设置的密码即可，如图 2-21 所示。需要注意：修改 neo4j 密码要牢记，因为在以后的程序开发等操作中还需要用到它。

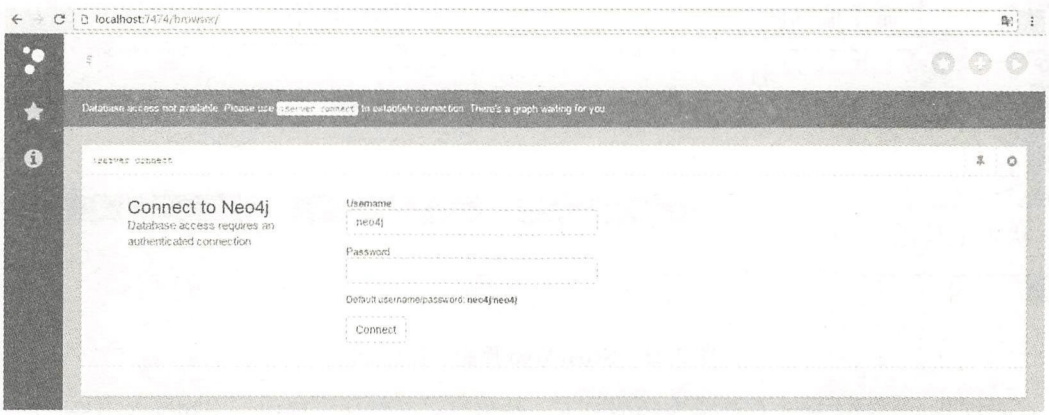


图 2-21　Neo4j Web 控制台的登录界面

修改完密码后，就进入了 Neo4j 的操作界面，如图 2-22 所示。

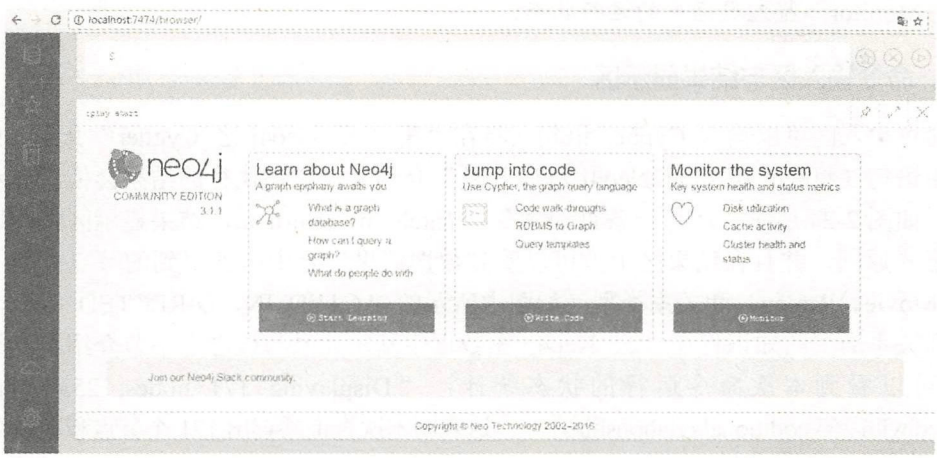


图 2-22　Neo4j Web 控制台登录后的界面

2.2 Neo4j 管理平台的使用

2.2.1 Web 管理平台的使用

Neo4j 采用 Web 网页作为管理平台的界面，并不需要安装其他操控软件。这样做的好处很明显：只要用浏览器就可以从任何电脑连接到数据库并进行相应操作。下面介绍 Neo4j Web 管理平台的使用方法。

当 Neo4j 安装完成并用 Web 端重设密码后，就可以看到 Neo4j 的操作界面了，如图 2-23 所示。操作界面主要由命令输入区、结果显示区、状态工具栏三部分组成。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

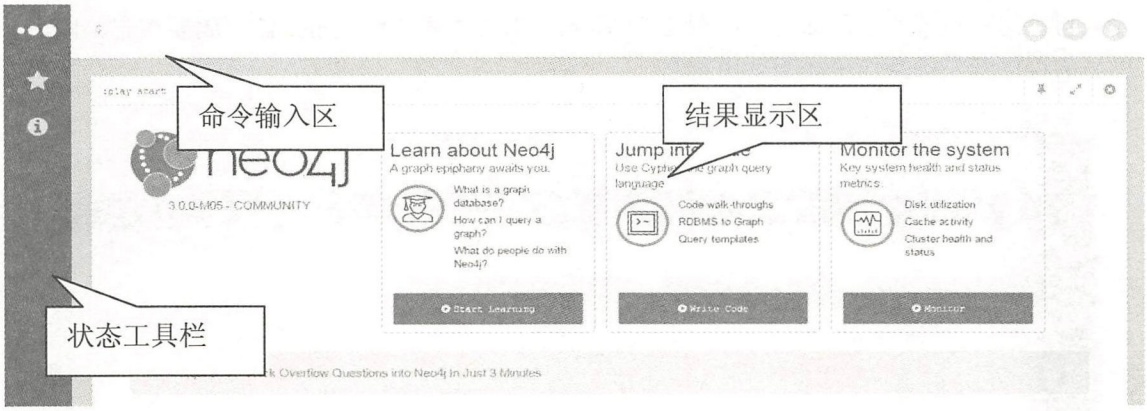


图 2-23　Neo4j Web 控制台各功能区

在刚打开页面但未输入任何命令的情况下，结果显示区默认呈现出一个初学者引导界面：

- Start Learning：了解 Neo4j 的基本概念。
- Write Code：创建官方引导实例。
- Monitor：监控数据库的运行状态。

2.2.1.1　命令输入区与结果显示区

在命令输入区可以输入 Cypher 语句（将在“第 3 章 Neo4j 之 Cypher”介绍）或 REST API 调用语句（将在“第 4 章 Neo4j 程序开发”介绍），语句执行的结果会依次显示在结果显示区。如图 2-24 所示，运行一条查询命令：match (n) return n;，结果显示区将会以节点、关系图形式返回，并且在结果区上方可以看到数据结果集合中所使用到的节点类型（如图中所示的 Movie、Person）和关系类型（如图中所示的 ACTED_IN、DIRECTED 等），关于节点类型和关系类型的知识将在“2.3 Neo4j 图数据中基本元素与概念”小节介绍。在结果区的下方还可以看到本次命令运行的状态统计：“Displaying 171 nodes, 253 relationships (completed with 253 additional relationships).”，这表明本次查询共查出 171 个节点和 253 个关系。

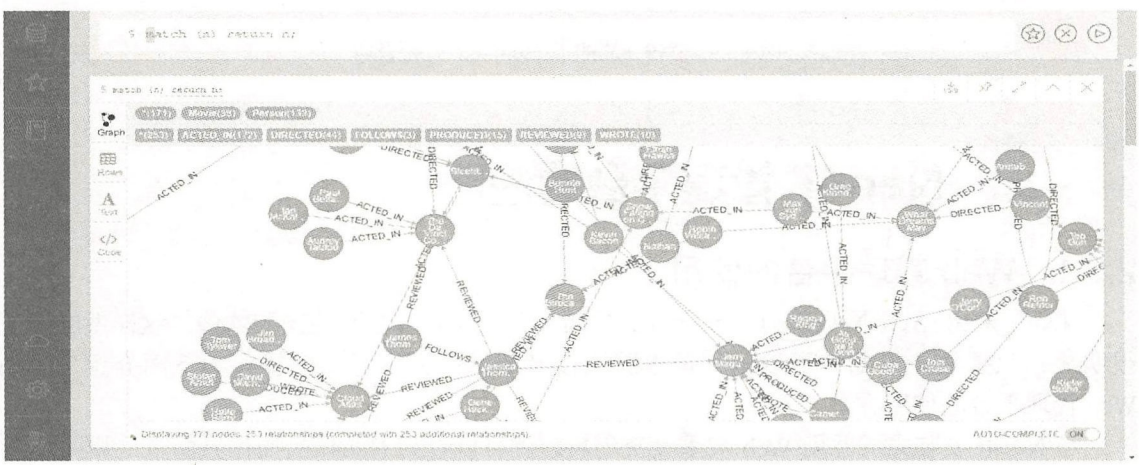


图 2-24　结果显示区

仅供非商业用途或交流学习使用

当然，也可以选择结果显示区左侧的 Rows、Text、Code 选项来以数据行、文本、JSON 格式返回，如图 2-25 所示。

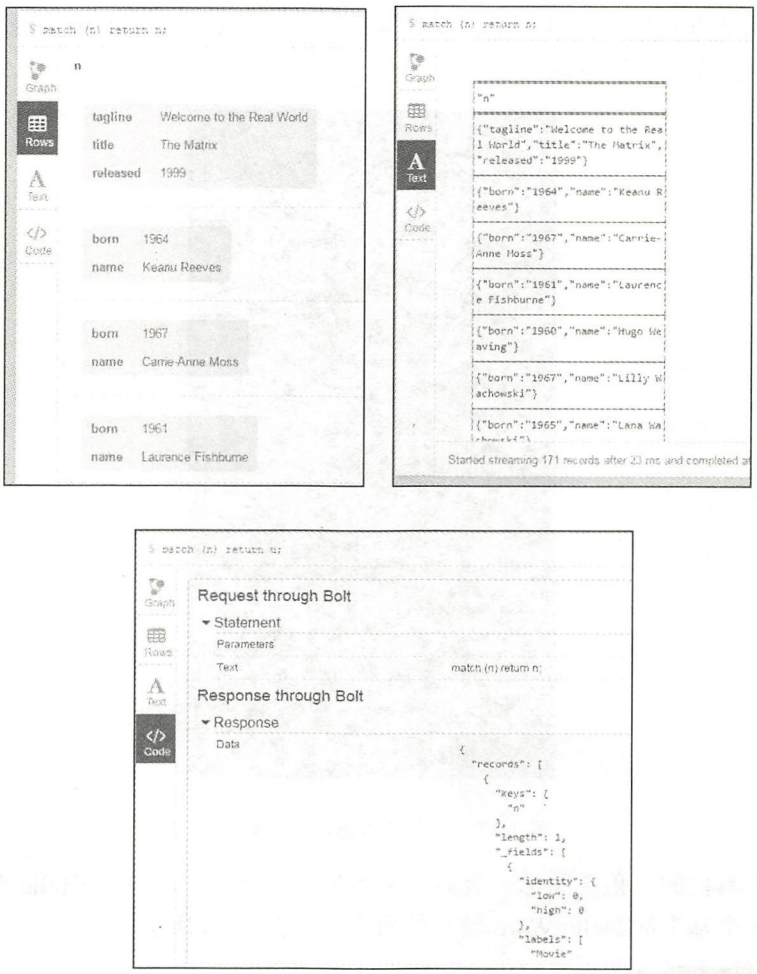


图 2-25　结果显示区选项卡菜单

如果使用的命令在运行中发生任何错误，结果区会显示相应的错误提示，如图 2-26 所示，在命令中输入错误的变量名并运行后结果区提示语法错误。

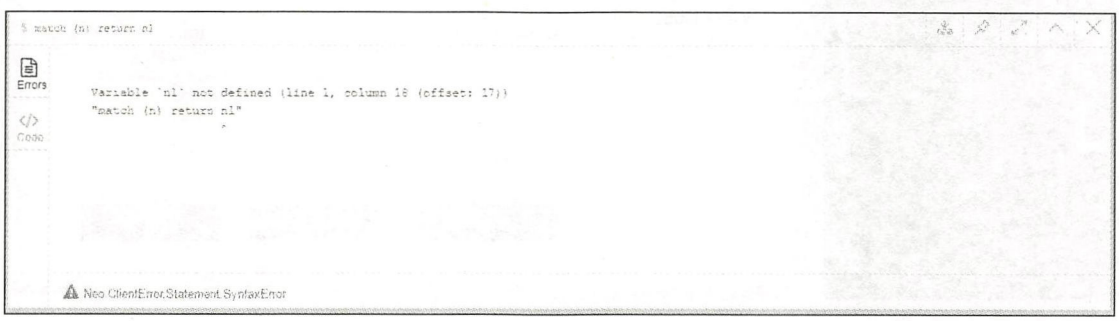


图 2-26　结果显示区的语法错误提示

仅供非商业用途或交流学习使用

2.2.1.2 状态工具栏

在状态工具栏单击数据库图标，如图 2-27 所示，可以查看到当前数据库的基本状态，如数据库中现有的节点类型、关系类型、属性名以及当前连接数据库的用户名和数据库版本、数据文件名、数据库大小等信息。

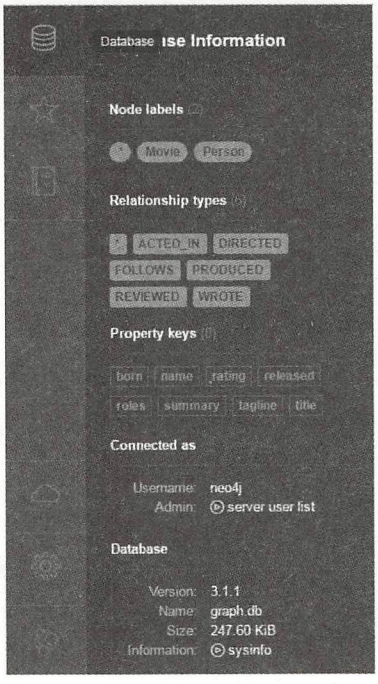


图 2-27　工具栏数据库信息选项卡

单击状态工具栏的五角星图标，会显示常用的一些命令，如单击 Hello Word，命令输入区会显示创建一个基本的 Hello Word 节点的命令，如图 2-28 所示。

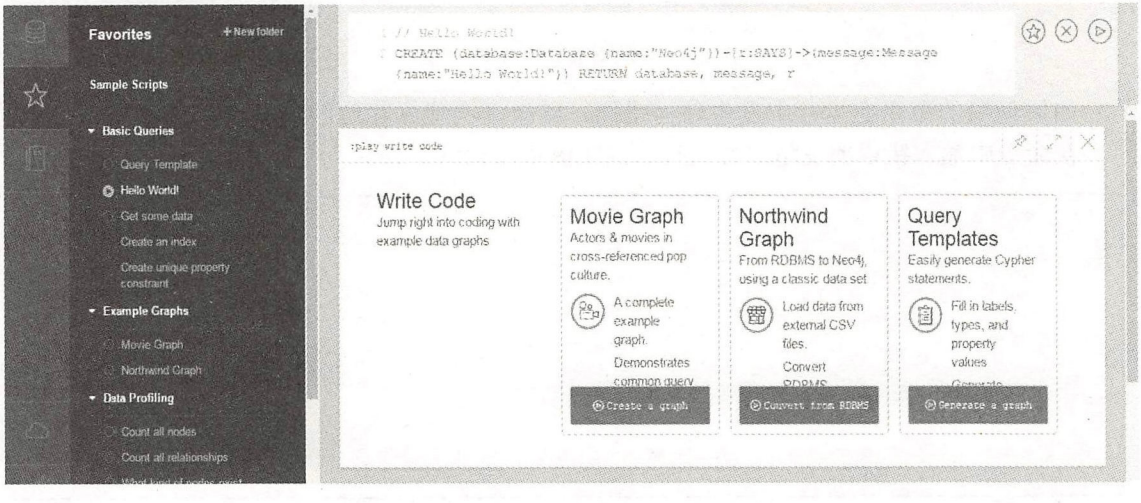


图 2-28　结果区帮助功能选项卡

仅供非商业用途或交流学习使用

在状态工具栏左侧另外还有 Document、Cloud Service、Browser Setting、About Neo4j，其中：

- Document 提供了官方文档的链接，用户可以单击跳转到官方文档。
- Cloud Service 提供了一些云备份的功能，用户可以将本地数据备份到官方云端。
- Browser Setting 用于设置 Web 操作界面的样式、布局。
- About Neo4j 给出了 Neo4j 官方的相关声明。

2.2.2　neo4j-shell 的使用

某些版本的 Linux 操作系统不带有可视化的操作界面（如服务器版本的 CentOS、Ubuntu 等），在本地无法使用浏览器打开 Neo4j Web 管理平台。为此，Neo4j 提供了一个可以在命令行状态下运行命令、返回结果的工具：neo4j-shell。

neo4j-shell 是官方自带的命令行工具，因此在安装完 Neo4j 后，neo4j-shell 就已经安装好了。并且 neo4j-shell 在各个不同操作系统下的 Neo4j 版本中都可找到。

2.2.2.1　启动 neo4j-shell

1. Windows 系统下 zip 解压版 Neo4j 启动 neo4j-shell

在 Windows 操作系统下，以管理员身份运行命令提示符并导航到 Neo4j 安装路径的 bin 文件夹，就可以看到 neo4j-shell.bat 文件，如图 2-29 所示。

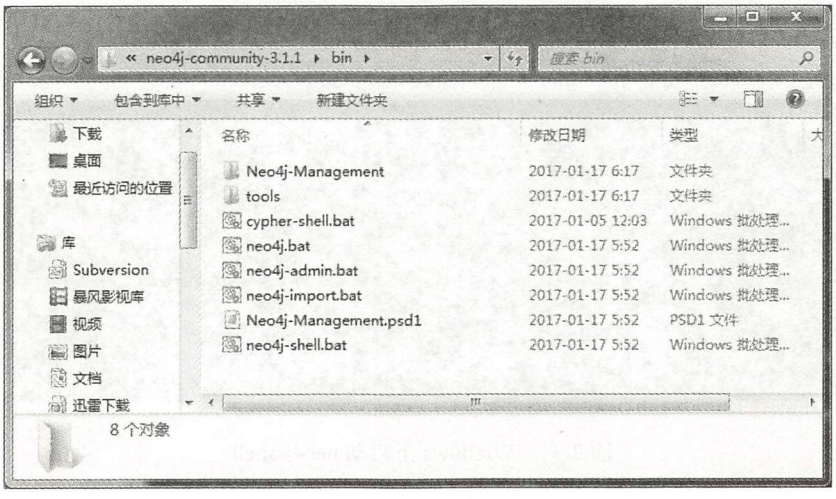


图 2-29　Windows 系统下 neo4j-shell 命令工具所在目录

直接运行 neo4j-shell 命令，可以得到此命令的所有参数，如图 2-30 所示。

仅供非商业用途或交流学习使用

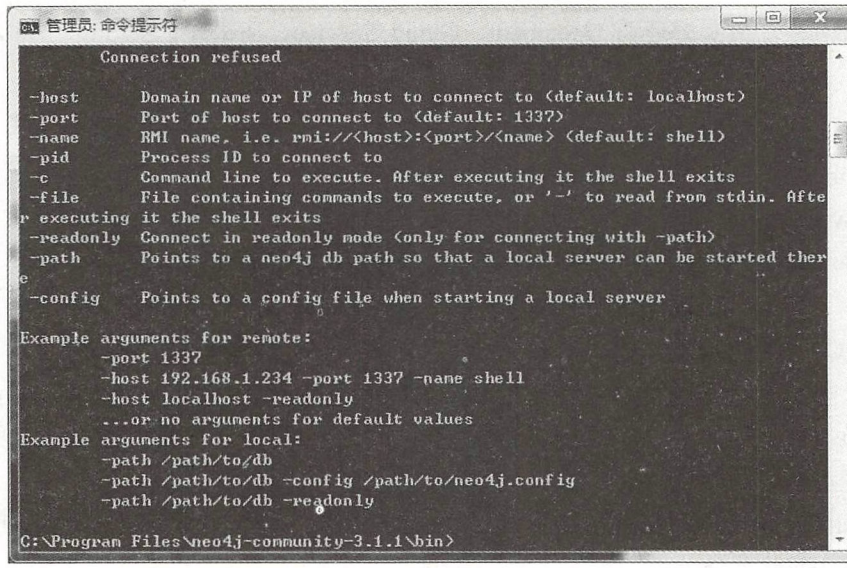


图 2-30　Windows 下 neo4j-shell 命令参数

如果要使用 neo4j-shell 操作默认路径下的数据库，那么直接运行 neo4j-shell 命令即可；如果需要打开指定路径下的 Neo4j 数据库，那么需要用–path 参数来指定数据文件路径，如运行下面命令（注意-path 参数采用相对路径）：

```
neo4j-shell -path ..\data\databases\graph.db
```

控制台出现 neo4j-sh (?)$ 提示符后，就可以使用 neo4j-shell 了，如图 2-31 所示。

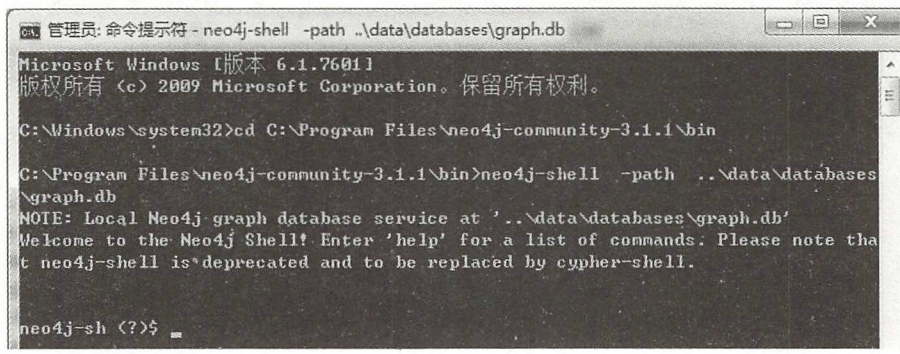


图 2-31　Windows 下启动 neo4j-shell

2. Windows 系统下安装版 Neo4j 启动 neo4j-shell

在安装版本的 Neo4j 运行目录的 bin 下，可以看到一个 neo4j-desktop-x.x.x.jar 文件，通过这个文件就可以启动 neo4j-shell 了，命令如下：

```
java -classpath bin\neo4j-desktop-1.9.4.jar org.neo4j.shell.StartClient
```

3. Linux 系统下启动 neo4j-shell

在 Linux 系统下，neo4j-shell 被安装在 Neo4j 运行目录 bin 文件下，如图 2-32 所示。

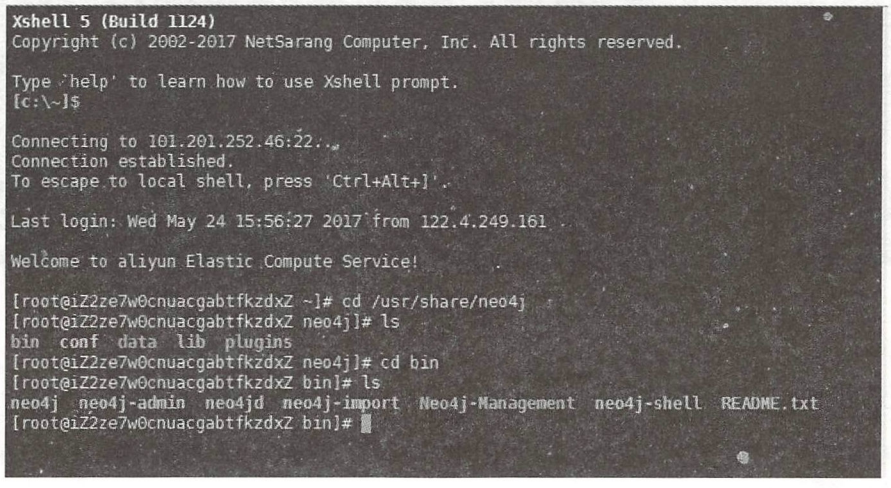


图 2-32　Linux 下 neo4j-shell 所在目录

只需要导航到 Neo4j 的安装目录的 bin 文件夹（默认情况下是/usr/share/neo4j/bin），然后运行 ./neo4j-shell 即可，如图 2-33 所示。

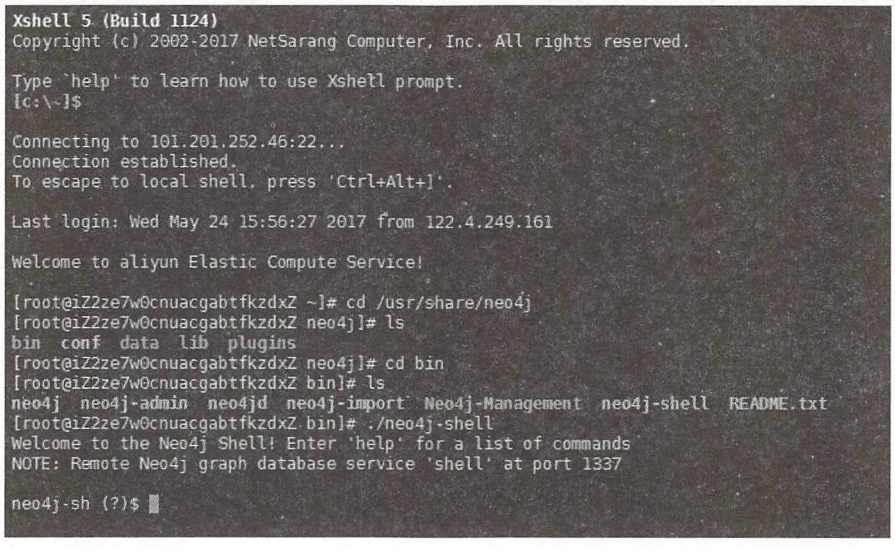


图 2-33　Linux 系统下运行 neo4j-shell

出现 neo4j-sh (?)$ 提示符后，就可以使用 neo4j-shell 了。

4. Mac 系统下启动 neo4j-shell

在 Mac 系统下，首先导航到 Neo4j 系统目录的 bin 目录下，然后运行如下命令：

```
java -cp neo4j-desktop-3.1.1.jar org.neo4j.shell.StartClient "$@"
```

出现 neo4j-sh (?)$ 提示符后，就可以使用 neo4j-shell 了，如图 2-34 所示。

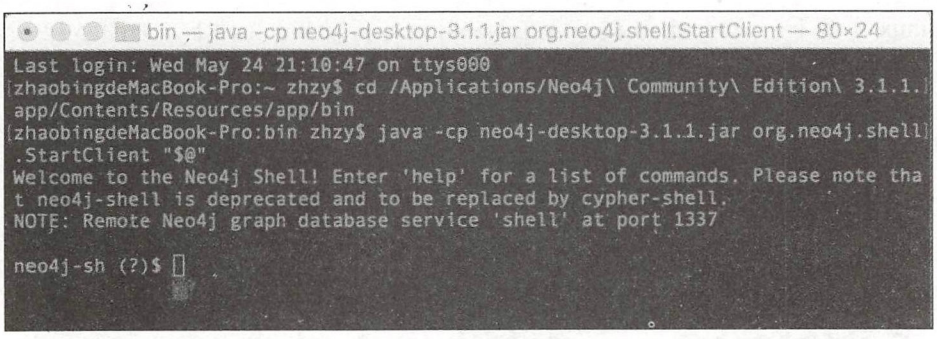


图 2-34　Mac 系统下运行 neo4j-shell 命令

2.2.2.2　使用 neo4j-shell

由于在任何操作系统中使用 neo4j-shell 的方式是一样的，所以不再分别介绍在不同操作系统下的使用方式。

在 neo4j-shell 命令提示符下直接输入 Cypher 命令（以；结束），然后按回车键运行即可。

比如使用 CREATE (n {name:"World"}) RETURN "hello", n.name 创建一个节点，然后再用 match (n) return n 查询出来。在 neo4j-shell 命令行中输入 quit，运行后退出 neo4j-shell，如图 2-35 所示。

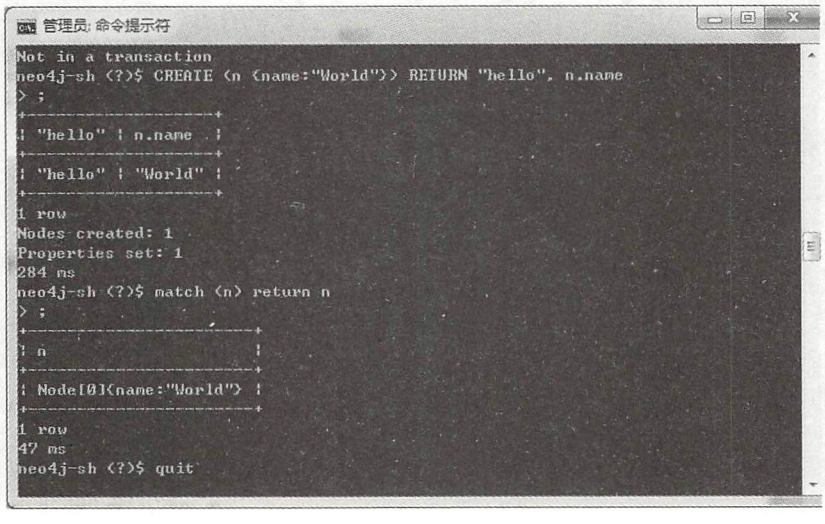


图 2-35　neo4j-shell 下运行 quit

2.3 Neo4j 图数据中基本元素与概念

2.3.1 节点

节点（Node）是图数据库中的一个基本元素，用以表示一个实体记录，就像关系数据库中的一条记录一样。在 Neo4j 中节点可以包含多个属性（Property）和多个标签（Label），如图 2-36 所示。

下面介绍一个最简单的节点，它只有一个属性，属性名是 name，属性值是 Tom，如图 2-37 所示。

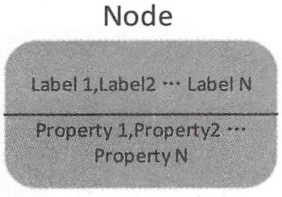


图 2-36　带有属性和标签的节点

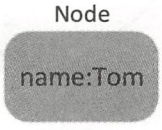


图 2-37　只有属性的简单节点

2.3.2 关系

关系（Relationship）同样是图数据库中的基本元素。当数据库中已经存在节点后，需要将节点连接起来构成图。关系就是用来连接两个节点，关系也称为图论的边（Edge），其始端和末端都必须是节点，关系不能指向空也不能从空发起。关系和节点一样可以包含多个属性，但关系只能有一个类型（Type），如图 2-38 所示；一个节点可以被多个关系指向或作为关系的起始节点，如图 2-39 所示多个关系指向同一节点。

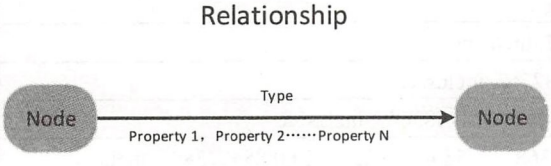


图 2-38　带有类型和属性的关系

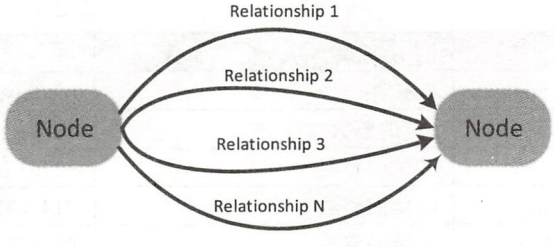


图 2-39　多个关系指向同一节点

关系必须有开始节点（Start node）和结束节点（End node），两头都不能为空，如图 2-40 所示。

节点可以被关系串联或并联起来，如图 2-41、图 2-42 所示。由于关系可以是有方向的，所以可在由节点、关系组成的图中进行遍历操作。

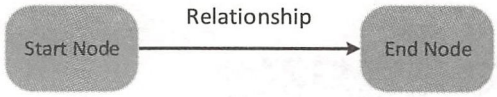


图 2-40　关系的起始节点和结束节点



图 2-41　关系串联节点

在图的遍历操作中我们可以指定关系遍历的方向或者指定为无方向，因此在创建关系时不必为两个节点创建相互指向的关系，而是在遍历时不指定遍历方向即可。

特别注意一个节点可以存在指向自己的关系，如图 2-43 所示。

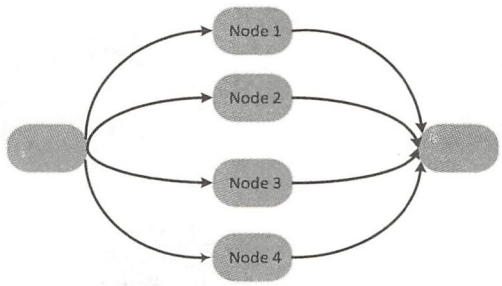


图 2-42　关系并联节点

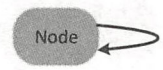


图 2-43　关系的起始、结束节点为同一节点

2.3.3　属性

上面提到节点和关系都可以有多个属性。属性是由键值对组成的，就像 Java 的哈希表一样，属性名类似变量名，属性值类似变量值。属性值可以是基本的数据类型，或者由基本数据类型组成的数组。

需要注意的是属性值没有 null 的概念，如果一个属性不需要了可以直接将整个键值对都移除，在使用 Cypher 或 Java API 时（详见“第 4 章 Neo4j 程序开发”），可用 IS NULL 关键字判断属性是否存在。表 2-1 列出了 Neo4j 中属性值的基本数据类型。

表 2-1　属性值类型

类型	说明	取值范围
boolean	布尔值	true/false
byte	8 位的整数	-128 to 127, inclusive
short	16 位的整数	-32768 to 32767, inclusive
int	32 位的整数	-2147483648 to 2147483647, inclusive
long	64 位的整数	-9223372036854775808 to 9223372036854775807, inclusive
float	32 位 IEEE 754 标准浮点数	
double	64 位 IEEE 754 标准浮点数	
char	16 位无符号整数代表的字符	u0000 to uffff (0 to 65535)
string	Unicode 字符序列	

2.3.4　路径

当使用节点和关系创建了一个图后，在此图中任意两个节点间都是可能存在路径的，如

图 2-44 所示。图中任意两节点都存在由节点和关系组成的路径，路径也有长度的概念，也就是路径中关系的条数。

当然也可以说单独一个节点就可以组成长度为 0 的路径，如图 2-45 所示。

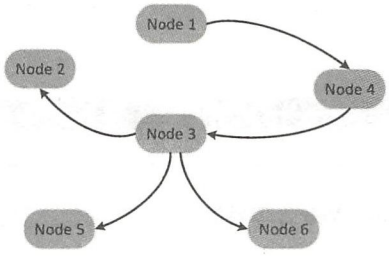


图 2-44　路径



图 2-45　长度为 0 的路径

如果是两个简单的节点，中间只存在一条关系，那这条路径的长度就是 1，如图 2-46 所示。



图 2-46　长度为 1 的路径

2.3.5　遍历（Traversal）

遍历一张图就是按照一定的规则，根据它们之间的关系，依次访问所有相关联的节点的操作。

对于遍历操作不必自己实现，因为 Neo4j 提供了一套高效的遍历 API，可以指定遍历规则，然后让 Neo4j 自动按照遍历规则遍历并返回遍历的结果。遍历规则可以是广度优先，也可以是深度优先。

想对遍历框架有一个深入的了解，请参见“第 3 章 Neo4j 之 Cypher”的有关章节；想了解更多的 Java 代码范例，请参见“第 4 章 Neo4j 程序开发”的有关章节。

2.4　官方入门实例介绍

为了方便读者入门，Neo4j Web 管理界面提供了一个官方入门实例“电影关系图”，帮助初学者在自己电脑上一步步创建一个入门级别的图数据结构。本节将围绕这个“电影关系图”实例一步步讲解、分析其创建和查询等操作。

首先，打开 Neo4j Web 管理界面（如图 2-47 所示）后，在引导实例区单击 Write Code 链接进入代码书写引导页，然后单击 Movie Graph 下的 Create a graph 链接就进入“电影关系图”实例引导界面了，如图 2-48 所示。

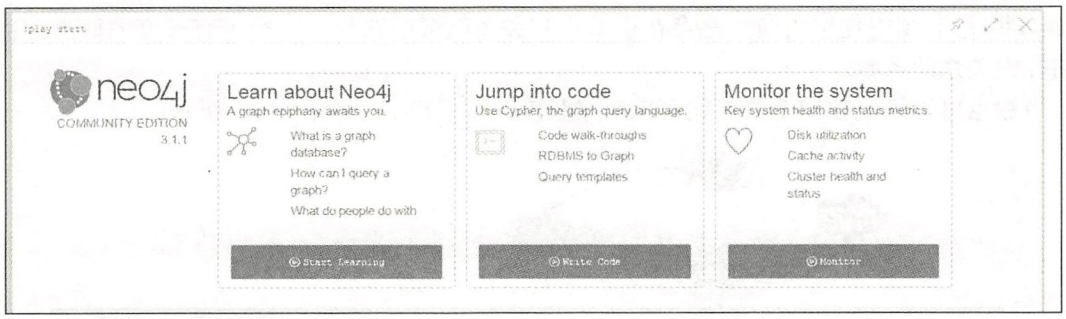


图 2-47　Neo4j Web 管理界面

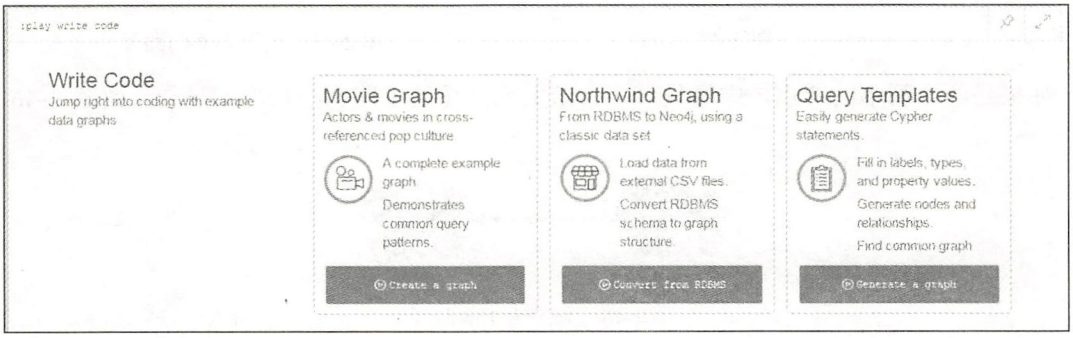


图 2-48　引导实例区

“电影关系图”实例将电影、电影导演、演员之间的复杂网状关系作为蓝本，使用 Neo4j 创建三者关系的图结构，虽然实例数据规模小但结构是相对完整的。

这个实例将指引读者学习以下入门操作：

（1）创建图数据：将电影、演员、导演等图数据导入到 Neo4j 数据库中。
（2）检索节点：检索特定电影和演员。
（3）查询关系：发现相关的演员和导演。
（4）查询关系路径：查询他们之间的关系路径。

2.4.1　创建图数据

单击“电影关系图”实例引导页的右侧换页箭头可以看到一些实例代码，其中包含多个带有 CREATE 关键字的 Cypher 语句。单击代码块，代码块将自动填入到命令行输入区。单击运行命令按钮，图数据就创建完成了。

下面，对这些创建语句进行分析说明，先不必完全弄懂这些命令，只需要了解它们的目的即可，关于这些 Cypher 语句的具体学习，在后续章节会有详细介绍。

【程序 2-1】创建电影节点

```
CREATE (TheMatrix:Movie {title:'The Matrix',
released:1999, tagline:'Welcome to the Real World'})
```

上面的 Cypher 语句使用 CREATE 指令创建了一个 Movie 节点，这个节点上带有三个属性{title:'The Matrix', released:1999, tagline:'Welcome to the Real World'}，分别表示这个电影的标题：The Matrix、发布时间：1999、宣传词：Welcome to the Real World。

上述 Cypher 语句运行后将会在数据库中创建一个 Movie 节点，在数据库中的存储形态如图 2-49 所示。



图 2-49　Movie 节点

【程序 2-2】创建人物节点

```
CREATE (Keanu:Person {name:'Keanu Reeves', born:1964})
```

上面代码使用 CREATE 指令创建了一个 Person 节点，节点带有两个属性{name:'Keanu Reeves', born:1964}。

在后续的 6 行代码中都使用了同样的 CREATE 指令分别创建了人物：Carrie、Laurence、Hugo、LillyW、LanaW 和 JoelS。

【程序 2-3】创建演员、导演关系

```
CREATE
  (Keanu)-[:ACTED_IN {roles:['Neo']}]->(TheMatrix),
  (Carrie)-[:ACTED_IN {roles:['Trinity']}]->(TheMatrix),
  (Laurence)-[:ACTED_IN {roles:['Morpheus']}]->(TheMatrix),
  (Hugo)-[:ACTED_IN {roles:['Agent Smith']}]->(TheMatrix),
  (LillyW)-[:DIRECTED]->(TheMatrix),
  (LanaW)-[:DIRECTED]->(TheMatrix),
  (JoelS)-[:PRODUCED]->(TheMatrix)
```

上面代码中除了使用 CREATE 指令外，还使用了箭头运算符，如：(Keanu)-[:ACTED_IN {roles:['Neo']}]->(TheMatrix)，这一行的意思是创建一个演员参演电影的关系，演员 Keanu 以角色 roles:['Neo']参演 ACTED_IN 到电影 TheMatrix 中。代码前 4 行都是创建演员参演电影关系的指令。

第 5 行指令：(LillyW)-[:DIRECTED]->(TheMatrix)，意思是创建导演与电影的关系，即 LillyW 导演了“[:DIRECTED]”电影 TheMatrix。

上面的指令运行完后，数据库中会有以下存储形态，如图 2-50 所示。

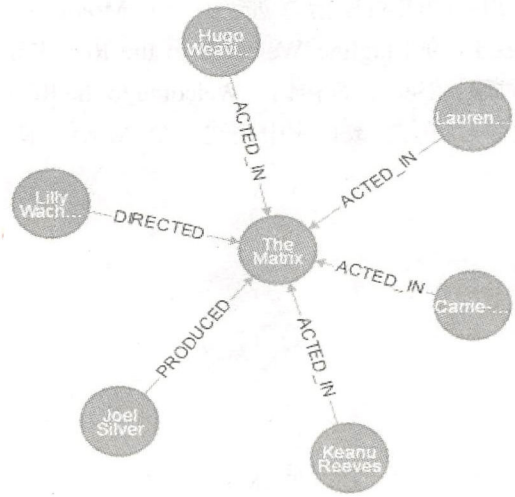


图 2-50　导演、电影关系图

这样数据库中一个电影、演员、导演、制片商的关系就创建出来了。在后面的代码中用了同样的指令分别创建了电影：The Matrix Revolutions、The Devil's Advocate、A Few Good Men、Top Gun、Jerry Maguire 等，然后又创建了与这些电影相关的演员、导演、制片商与他们之间的关系。

通过上述的创建指令就把“电影关系图”实例创建完成了。

2.4.2　检索节点

图数据结构创建完毕后，下面介绍检索节点的相关操作。

2.4.2.1　查找人员

【程序 2-4】查找名为“Tom Hanks”的人物

```
MATCH (tom {name: "Tom Hanks"}) RETURN tom
```

上面语句使用 MATCH 指令查找匹配条件：{name: "Tom Hanks"}的节点，执行的结果如图 2-51 所示。

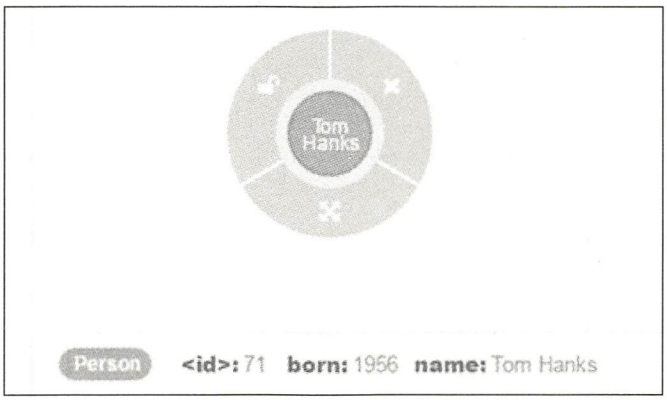


图 2-51　查找到的节点

可以在结果显示区看到查询到的节点，注意在结果展示区单击节点图标，会查看到节点的属性。

2.4.2.2　查找电影节点

【程序 2-5】查找名为“Cloud Atlas”的电影

```
MATCH (cloudAtlas {title: "Cloud Atlas"}) RETURN cloudAtlas
```

上面指令查找匹配条件{title: "Cloud Atlas"}的节点，返回结果如图 2-52 所示。

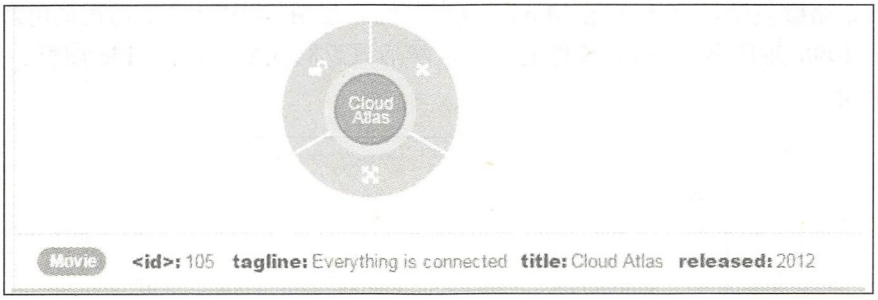


图 2-52　查找到的节点

2.4.2.3　随机查找多个人物的人名

【程序 2-6】随机查找 10 个人物的人名

```
MATCH (people:Person) RETURN people.name LIMIT 10
```

上面指令查找 10 个 Person 节点，然后返回每个节点的 name 属性，返回结果如图 2-53 所示。

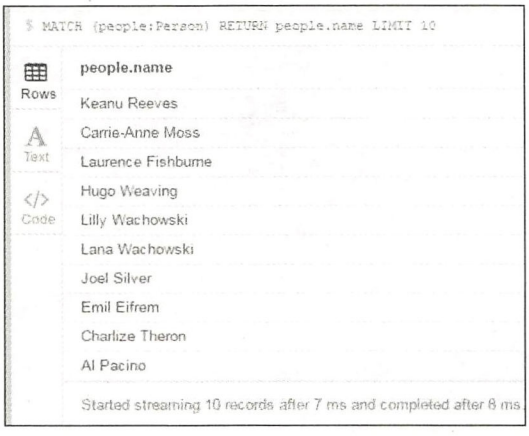


图 2-53　name 属性列表

在本次结果中，由于只返回 name 属性，所以就不用图形化的形式返回了。

2.4.2.4　查找多个电影

【程序 2-7】查找 1990 年到 2000 年发行的电影的名称

```
MATCH (nineties:Movie)
WHERE nineties.released > 1990 AND nineties.released < 2000
RETURN nineties.title
```

上面指令略微复杂，首先匹配 Movie 节点，然后使用 WHERE 子句查询电影的 released 属性值大于 1990 并且小于 2000 条件的节点，然后只返回匹配节点的 title 属性。返回结果为如图 2-54 所示。

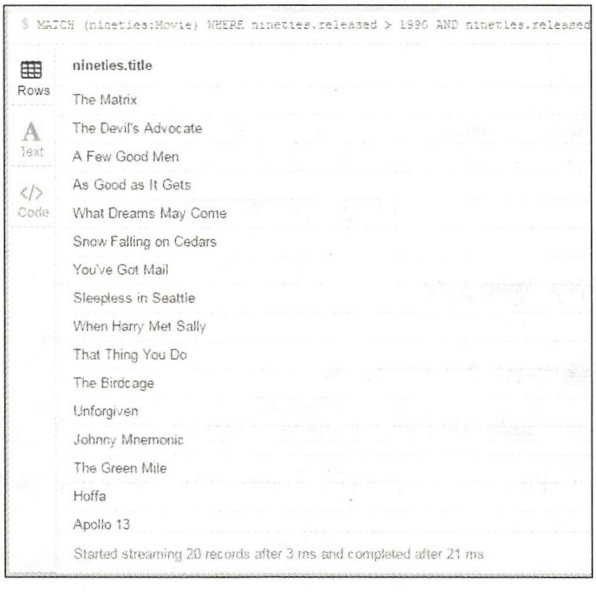


图 2-54　title 属性列表

2.4.3　查询关系

下面拓展 MATCH 指令的使用。

2.4.3.1　查找演员参演的电影

【程序 2-8】查找“Tom Hanks”参演过的电影的名称

```
MATCH (tom:Person {name: "Tom Hanks"})-[:ACTED_IN]->(tomHanksMovies) RETURN
tom,tomHanksMovies
```

上述指令首先匹配节点类型为 Person、属性为{name: "Tom Hanks"}的节点，然后匹配此节点具有关系[:ACTED_IN]，并且此关系指向某个电影节点的节点。返回结果如图 2-55 所示。



图 2-55　“Tom Hanks”参演过的电影

通过结果可以看到演员 Tom Hanks 参演过的所有电影。

【程序 2-9】查找谁导演了电影“Cloud Atlas”

```
MATCH (cloudAtlas {title: "Cloud Atlas"})<-[:DIRECTED]-(directors)
RETURN directors.name
```

上面指令首先匹配属性为{title: "Cloud Atlas"}的节点，然后匹配此节点具有关系[:DIRECTED]并且是被某个节点指向的节点，再返回匹配节点的 name 属性。返回结果如图 2-56 所示。



图 2-56　电影 Cloud Atlas 的导演名单

通过结果可以看到 Lily Wachowski、Lana Wachowski、Tom Tykwer 导演了电影 Cloud Atlas。

【程序 2-10】查找与 Tom Hanks 同出演过电影的人

```
MATCH (tom:Person {name:"  m Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-
    (coActors)
RETURN coActors.name
```

上面指令首先匹配节点类型为 Person、属性为{name:"Tom Hanks"}的节点，然后匹配此节点通过[:ACTED_IN]关系指向的节点 m，并且同时匹配某个节点 coActors 也通过[:ACTED_IN]关系指向的节点 m，然后返回匹配节点 m 的 name 属性。返回结果如图 2-57 所示。

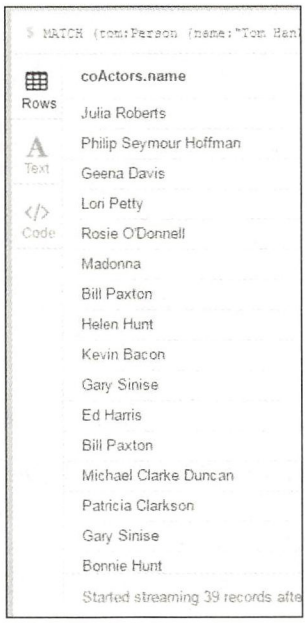


图 2-57　与“Tom Hanks”同出演过电影的人

这样与 Tom Hanks 同时出演过电影的人的姓名就查出来了。

【程序 2-11】查找与电影“Cloud Atlas”相关的所有人

```
MATCH (people:Person)-[relatedTo]-(:Movie {title: "Cloud Atlas"})
RETURN people.name, Type(relatedTo), relatedTo
```

上面指令首先匹配节点类型为 Person 的节点，然后匹配节点类型为 Movie、节点属性为{title: "Cloud Atlas"}的节点，最后匹配他们两者之间存在某种关系（无论是导演还是演员关系）的情况，然后将人名、电影的关系类型、电影的关系同时返回。返回结果如图 2-58 所示。

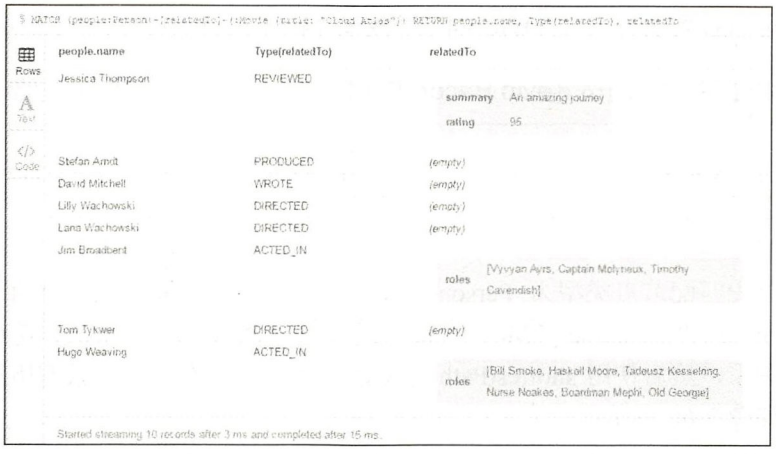


图 2-58　与电影“Cloud Atlas”相关的所有人

通过结果可以看到返回了三列数据，这与 RETURN 语句后面的 people.name、Type(relatedTo)、 relatedTo 相对应。

2.4.3.2　查询关系路径

您或许听说过“六度空间”理论，也就是说，世界上任何两个人，他们之间至多通过 6 条关系路径就可以相互联系到彼此。使用 Neo4j 的关系路径查询可以查找任意深度的关系路径，也就很轻松地能够实现人脉关系查询了。

【程序 2-12】查找与演员“Kevin Bacon”存在 4 条及以内关系的任何演员和电影

```
MATCH (bacon:Person {name:"Kevin Bacon"})-[*1..4]-(hollywood)
RETURN DISTINCT hollywood
```

上面指令首先匹配节点类型为 Person、属性为{name: " Kevin Bacon "}的节点，然后将关系深度限制为从 1 到 4 再进行遍历，最后返回匹配的所有节点。返回结果如图 2-59 所示。

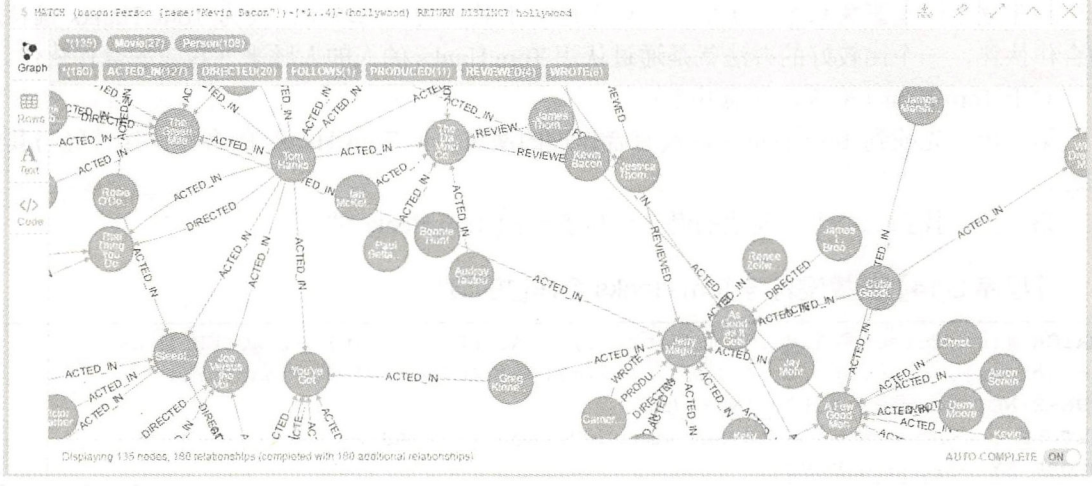


图 2-59　与演员 Kevin Bacon 存在 4 条及以内关系的演员和电影

通过结果可以看到演员 Kevin Bacon4 度关系以内的演员和电影网络是很庞大的。

【程序 2-13】查找与演员 Kevin Bacon 与 Meg Ryan 之间的最短关系路径

```
MATCH p=shortestPath(
(bacon:Person {name:"Kevin Bacon"})-[*]-(meg:Person {name:"Meg Ryan"})
)
RETURN p
```

上面指令首先匹配节点类型为 Person、属性为{name: "Kevin Bacon"}的节点，再匹配节点类型为 Person、属性为{name: " Meg Ryan "}的节点，两者用[*]关系操作符相连代表两者存在任意深度的关系，然后使用 shortestPath 方法返回两者在所有深度关系遍历路径中最短的一条。返回结果如图 2-60 所示。

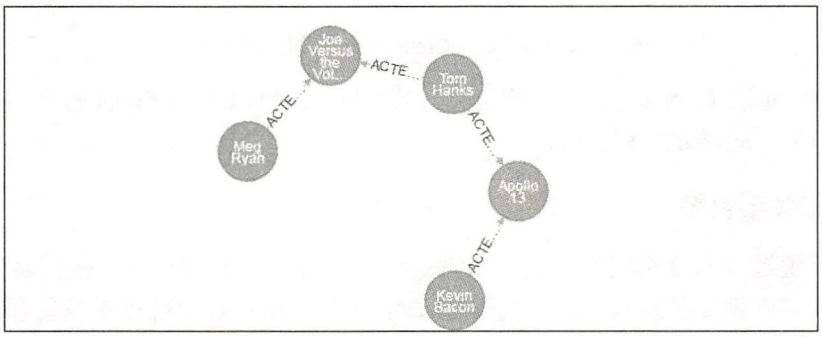


图 2-60 “Kevin Bacon”与“Meg Ryan”之间的最短关系路径

通过结果可以看到演员 Meg Ryan 与 Tom Hanks 同参演过 Joe Versus the Volcano 电影，而 Tom Hanks 与 Kevin Bacon 同参演过 Apollo 13 电影，这就是他们两者之间的最短关系路径。

2.4.4 思考与练习

基于这个“电影关系图”实例，可以考虑一下其他的应用场景：要为 Tom Hanks 推荐新的合作伙伴，一个比较好的办法就是通过认识 Tom Hanks 的人的人脉来寻找新的合作伙伴。

对于 Tom Hanks 来说，这意味着：

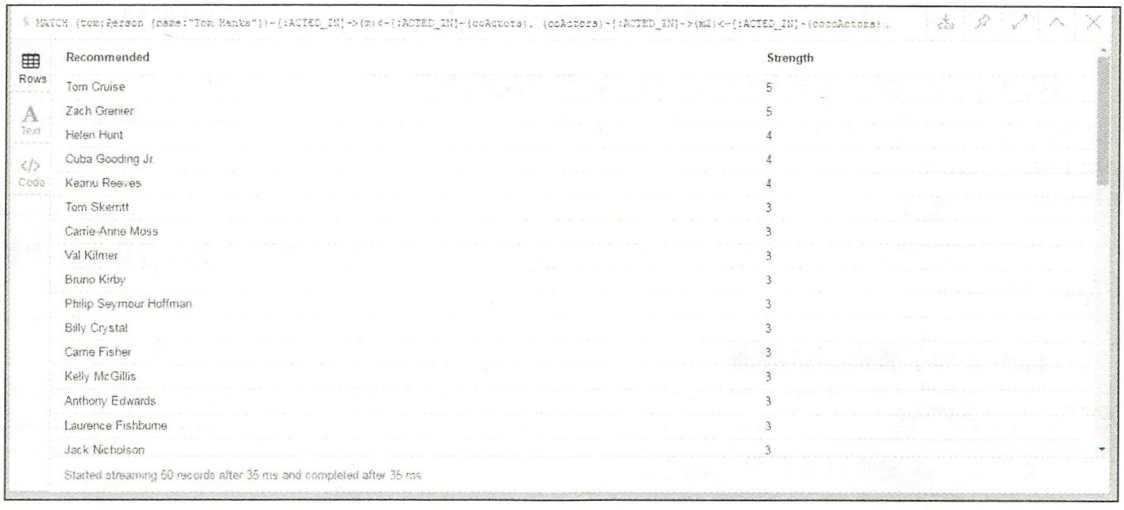
第一步，先找到 Tom Hanks 还没有合作过的演员，但 Tom Hanks 的合作伙伴曾经与其合作过。

第二步，找到一个可以向他的潜在合作者介绍 Tom Hanks 的人。

【程序 2-14】查找没有与 Tom Hanks 合作过的演员

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-
(coActors),(coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors)
WHERE NOT (tom)-[:ACTED_IN]->(m2)
RETURN cocoActors.name AS Recommended, count(*) AS Strength
ORDER BY Strength DESC
```

结果如图 2-61 所示。



Recommended	Strength
Tom Cruise	5
Zach Grenier	5
Helen Hunt	4
Cuba Gooding Jr.	4
Keanu Reeves	4
Tom Skerritt	3
Carrie-Anne Moss	3
Val Kilmer	3
Bruno Kirby	3
Philip Seymour Hoffman	3
Billy Crystal	3
Carrie Fisher	3
Kelly McGillis	3
Anthony Edwards	3
Laurence Fishburne	3
Jack Nicholson	3

Started streaming 50 records after 35 ms and completed after 35 ms

图 2-61　没有与 Tom Hanks 合作过的演员

【程序 2-15】找人将 Tom Hanks 介绍给 Tom Cruise

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-
(coActors), (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cruise:Person
{name:"Tom Cruise"})
RETURN tom, m, coActors, m2, cruise
```

结果如图 2-62 所示。

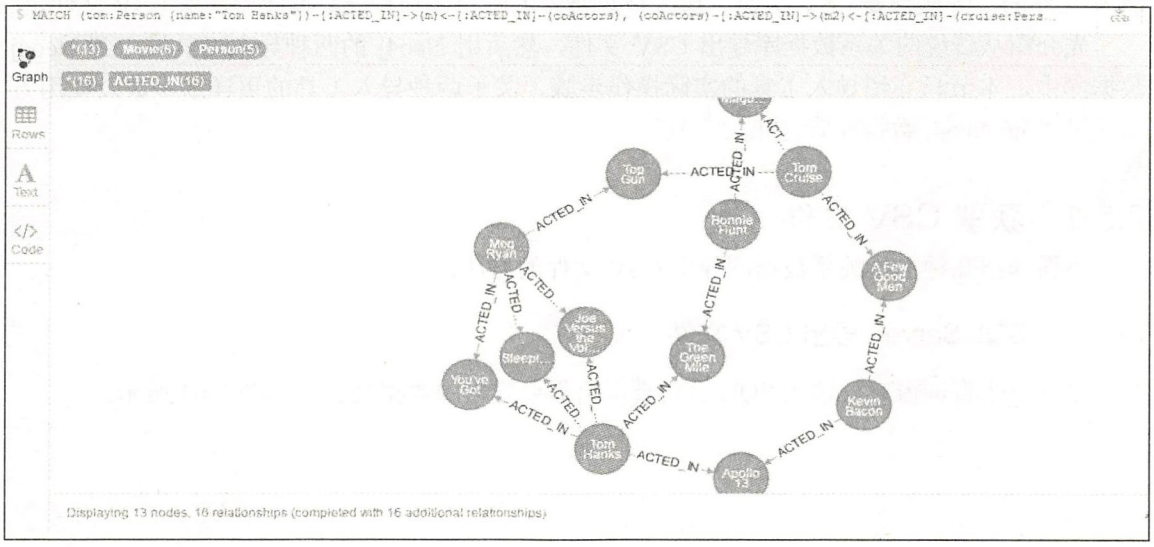


图 2-62　找人将 Tom Hanks 介绍给 Tom Cruise

2.4.5 清空数据库

上面的数据在操作完后，有时候需要清理掉这些数据。下面的代码就可以做到。

【程序 2-16】清空所有 Person、Movie 节点及其所有关系

```
MATCH (a:Person),(m:Movie)
OPTIONAL MATCH (a)-[r1]-(), (m)-[r2]-()
DELETE a,r1,m,r2
```

这样操作完成后，为了查询数据是否已经被清空了，可以运行以下命令先看数据库中是否还有任何数据。

【程序 2-17】查询任意数据

```
MATCH (n) RETURN n
```

如果没有，就说明已经删除成功了。

2.5 批量导入工具的使用

在初次使用 Neo4j 图数据库时，很多读者需要将自己使用的关系数据库或用其他存储形式存储的数据批量导入到 Neo4j 图数据库中。为此，本节将介绍怎样使用 Neo4j 导入工具来批量导入数据。

目前，导入到 Neo4j 的工具有两种：load csv 指令和 neo4j-import 工具，这两种都是基于 CSV 文件的，既然需要 CSV 文件，那么有必要先考虑如何从其他数据库中获取 CSV 文件。

先介绍从传统的关系数据库导出 CSV 文件，然后用 Neo4j 的两种导入工具导入到 Neo4j 数据库中，本节只介绍导入工具的实际操作步骤，关于两种导入工具的更详细参数及使用方法参见“5.6 Neo4j 数据库管理相关工具”一节。

2.5.1 获取 CSV 文件

下面介绍几种主流关系数据库导出 CSV 文件的方法。

2.5.1.1 SQL Server 导出 CSV 文件

打开一个查询窗口，输入 SQL 语句查询出需要导出的数据列表，如图 2-63 所示。

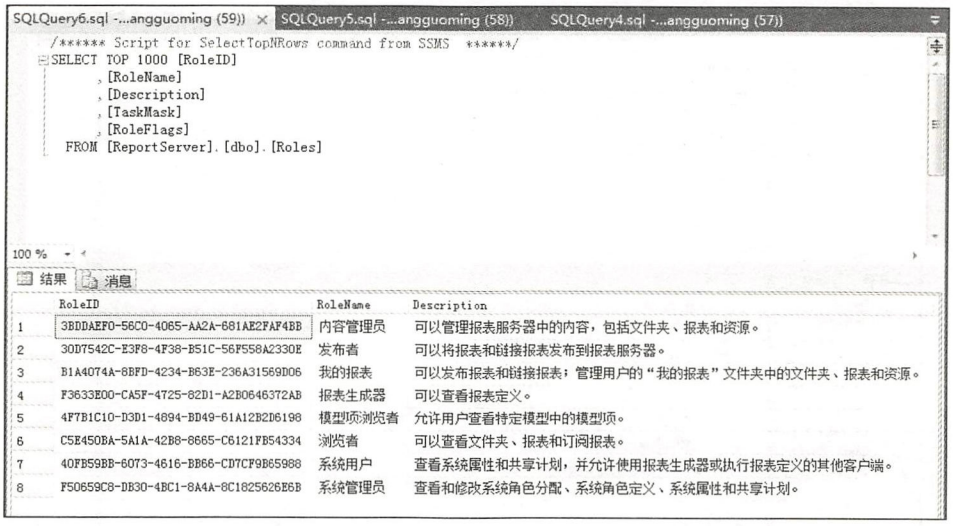


图 2-63　SQL 查询

在查询的结果栏目中，复制所有查询出来的数据。右击，在快捷菜单中选择“将结果另存为”，如图 2-64 所示。



图 2-64　查询结果另存为

这里可以选择保存为 CSV 文件或者是 txt 文件，我们选择 CSV 文件，如图 2-65 所示。用 Excel 打开 CSV 文件，就可以看到 CSV 文件的内容了。

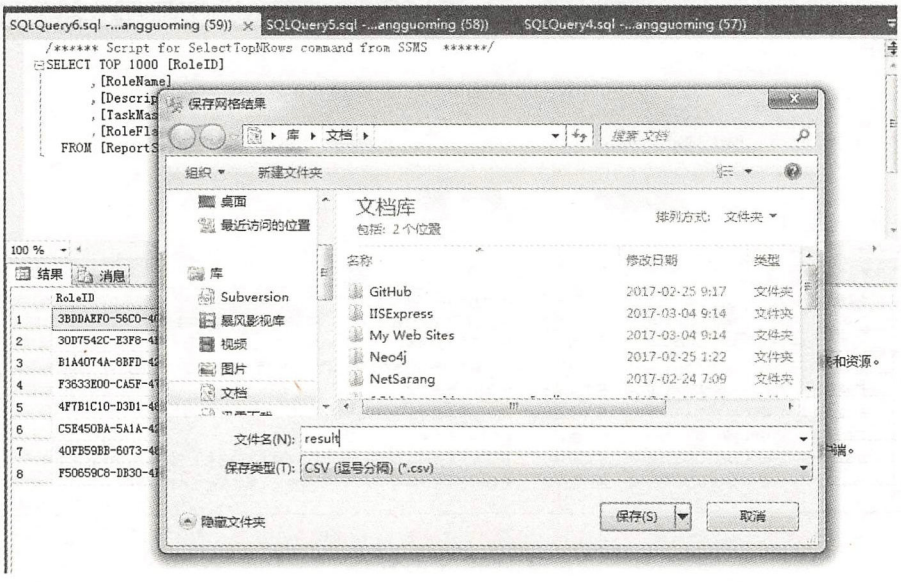


图 2-65　保存 CSV 文件

2.5.1.2　MySQL 导出 CSV 文件

MySQL 支持将查询结果直接导出为 CSV 文本格式，在使用 select 语句查询数据时，在语句后面加上导出指令即可，格式如下：

- into outfile <导出的目录和文件名>：指定导出的目录和文件名。
- fields terminated by <字段间分隔符>：定义字段间的分隔符。
- optionally enclosed by <字段包围符>：定义包围字段的字符（数值型字段无效）。
- lines terminated by <行间分隔符>：定义每行的分隔符。

下面举一个使用 MySQL 导出 CSV 文件的例子：

【程序 2-18】MySQL 导出 CSV 文件

```
select * from mydatatable where mytag like 'E1%'
into outfile 'E:\E1.csv' fields terminated by ',' optionally enclosed by '"'
lines terminated by '\r\n';
```

此条指令执行后，会把 mydatatable 表中部分匹配查询条件的记录导出到 E1.csv 文件中。其中每个字段以逗号分隔，字段内容是以双引号包围的字符串，每条记录使用\r\n 换行，如图 2-66 所示。

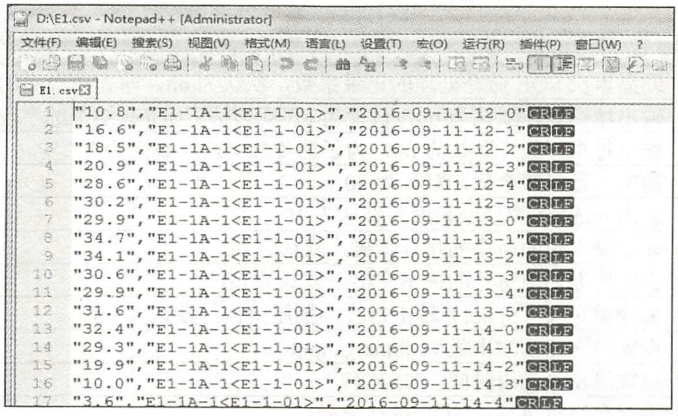


图 2-66　CSV 文件格式

2.5.1.3　Oracle 导出 CSV 文件

可以使用 sqlplus 导出 CSV 文件，这是比较方便的一种方式，当然读者也可以考虑采用其他方式。

首先创建 spool.sql 文件，内容如下：

【程序 2-19】创建 spool.sql 文件

```
set colsep ,
set feedback off
set heading off
set trimout on
spool E:\user.csv
select '"' || user_name || '","' || user_age || '","' || user_card || '","'
|| user_sex || '","' || user_addres || '","' || user_tel || '"' from user_
bl;
spool off
exit
```

然后，运行 sqlplus 命令：

```
sqlplus -s 用户名/密码@数据库名 @spool.sql
```

这样就从 Oracle 中将 lfc_ bl 表中的数据导出到 E:\lfc_ bl.csv 里面了。

对于 spool.sql 文件的指令详细解析请参考表 2-2 所示。

表 2-2　sqlplus 命令参照

命令	含义
set colsep' ';	域输出分隔符
set newp none	设置查询出来的数据分多少页显示，如果需要连续的数据，中间不要出现空行就把 newp 设置为 none，这样输出的数据行都是连续的，中间没有空行之类的
set echo off;	显示 start 启动的脚本中的每个 SQL 命令，默认为 on
set echo on;	设置运行命令时是否显示语句
set feedback on;	设置显示“已选择 XX 行”

（续表）

命令	含义
set feedback off;	回显本次 SQL 命令处理的记录条数，默认为 on，即去掉最后的"已经选择 10000 行"
set heading off;	输出域标题，默认为 on，设置为 off 就去掉了 select 结果的字段名，只显示数据
set pagesize 0;	输出每页行数，默认为 24，为了避免分页，可设定为 0
set linesize 80;	输出一行字符个数，默认为 80
set numwidth 12;	输出 number 类型域长度，默认为 10
set termout off;	显示脚本中的命令的执行结果，默认为 on
set trimout on;	去除标准输出每行的拖尾空格，默认为 off
set trimspool on;	去除重定向（Spool）输出每行的拖尾空格，默认为 off
set serveroutput on;	设置允许显示输出类似 dbms_output
set timing on;	设置显示“已用时间：XXXX”
set autotrace on-;	设置允许对执行的 SQL 进行分析

2.5.1.4 CSV 内容格式注意事项

Neo4j 对 CSV 文件的格式是有要求的，在讨论如何导入 CSV 之前，先把 CSV 文件中一些常见的错误列举出来。读者可以使用文本编辑器打开自己的 CSV 文件查看是否出现以下错误。

CSV 文件内容格式常见错误如下：

（1）在 CSV 文件开始处存在 BOM 字节顺序标记（2 个 UTF-8 字符），如果存在需要删除它们。

（2）文件内存在非文本类型的字符，如果存在就删除它们。

（3）存在不规则换行符，如：混合 Windows 和 UNIX 换行符，如果有需要确保它们一致，最好选择 UNIX 风格。

（4）CSV 文件头与数据不一致（内容相比头缺少列或多出列，头中有不同的分隔符），如有此情况需要修复头部。

（5）带引号和不带引号的文本字段中出现换行符，如存在就删除换行符。

（6）存在杂散的引号，非文本中存在独立双引号或单引号，如存在就转义或删除杂散引号。

2.5.2 使用 Load CSV 指令导入到 Neo4j

Neo4j 提供了 Load CSV 命令帮助我们将 CSV 数据文件导入到 Neo4j 中，下面给出几个读取 CSV 文件，但不存入数据库的例子。

2.5.2.1 简单导入 CSV 数据

Load CSV 读取但不存入数据库：

```
// 查看前 CSV 文件行数
LOAD CSV FROM "file-url" AS line
RETURN count(*);
// 查看前 CSV 文件前5行
LOAD CSV FROM "file-url" AS line WITH line
RETURN line
LIMIT 5;
//查看前 CSV 文件，并带有头部数据
```

```
LOAD CSV WITH HEADERS FROM "file-url" AS line WITH line
RETURN line
LIMIT 5;
```

上述例子仅仅是用来读取 CSV 文件，它并没有将数据存入到数据库中。下面介绍 Load CSV 的用法。

LOAD CSV FROM "file-url" AS line，这条指令就是将指定路径下的 CSV 文件读取出来，其中"file-url"就是文件的地址，可以是本地文件路径也可以是网址，只要能从地址中读取到 CSV 文件即可，因此也可以这样写：

```
LOAD CSV FROM 'http://we-yun.com/neo4jguide/movie.csv' AS line
RETURN line
```

这样就可以读取网址指定的 movie.csv 文件。

或者可以使用本地文件路径：

```
LOAD CSV FROM ' file:///E:/products.csv' AS line
RETURN line
```

这样就可以读取到 E:/products.csv 文件。

如果把 CSV 文件放置在 Neo4j 系统路径的 import 文件夹内，就只不需要指定 CSV 文件的路径，如下所示，可以读取到放入 import 文件夹内的 products.csv 文件。

```
LOAD CSV FROM ' file:/// products.csv' AS line
RETURN line
```

对于 Return 语句，它是用来返回并显示结果到结果显示区的语句。

对于 Limit 语句，是用来限制返回的行数。

这样就可以读取 CSV 文件了，但它并没有存入到数据库中，要导入到 Neo4j 数据库中需要使用相应的 Create 语句，下例可以读取网址指定的 movie.csv 文件并批量创建节点 Movie，每个节点的 title、released 和 tagline 属性由导入的 CSV 文件的每行指定。

```
LOAD CSV FROM 'http://we-yun.com/neo4jguide/movie.csv' AS line
CREATE (:Movie { title: line[0] , released: line[1] , tagline: line[2]})
```

执行完上面语句后，会看到结果显示区显示了所创建的节点数量，如图 2-67 所示。

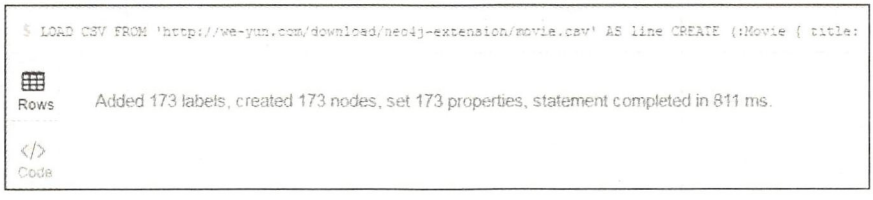


图 2-67　显示区显示了所创建的节点数量

然后，用以下语句来查看下数据库内是否已经有导入的数据了：

```
MATCH (n:Movie) RETURN n
```

运行上面语句后，可以得到如图 2-68 所示的结果。

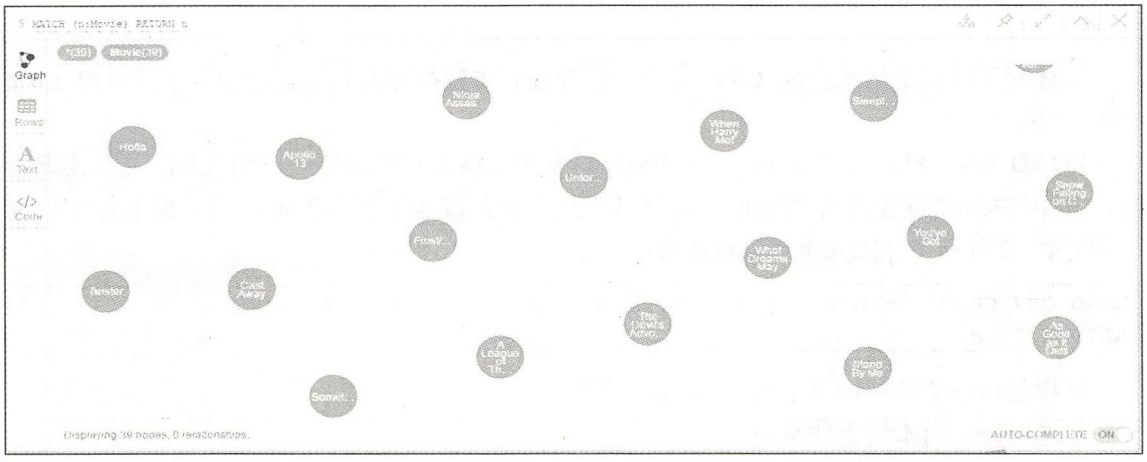


图 2-68　查询导入后结果

如上图所示，这说明已经导入数据成功了。

2.5.2.2　导入 CSV 时附带表头

下例介绍如何在导入 CSV 时附带上表头。

```
LOAD CSV WITH HEADERS FROM 'http://we-yun.com/neo4jguide/tracks.csv ' AS line
CREATE (:Track { TrackId: line.Id, Name: line.Track, Length: line.Length})
```

上述语句添加了 WITH　HEADERS 子句，它的功能就是在导入 CSV 时附带上头部，头部由 line.Id、line.Track 指定。执行完上面语句后，可使用下面语句查看一下：

```
MATCH (n:Track) RETURN n
```

将得到如图 2-69 所示的结果。

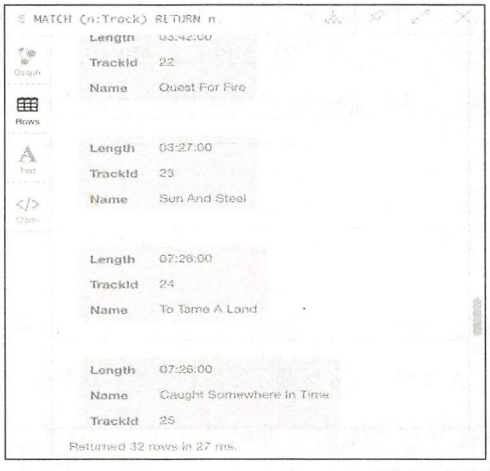


图 2-69　查询导入后结果

可以看到，每个数据元素都带有 TrackId、 Name、Length 头部元素。

2.5.2.3　导入 CSV 大文件

如果要导入包含大量数据的 CSV 文件，则可以使用 PERODIC COMMIT 子句。

使用 PERIODIC COMMIT 指示 Neo4j 在执行完一定行数后提交数据再继续，这样减少了内存开销。

使用 PERIODIC COMMIT 默认值为 1000 行，因此数据将每一千行提交一次。

要使用 PERIODIC COMMIT，只需在 LOAD CSV 语句之前插入 USING PERIODIC COMMIT 语句。

具体使用方法如下所示。

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM 'http://we-yun.com/neo4jguide/tracks.csv' AS line
CREATE (:Track { TrackId: line.Id, Name: line.Track, Length: line.Length})
```

可以让其每 800 行提交一次，如下所示。

```
USING PERIODIC COMMIT 800
LOAD CSV WITH HEADERS FROM 'http://we-yun.com/neo4jguide/tracks.csv' AS line
CREATE (:Track { TrackId: line.Id, Name: line.Track, Length: line.Length})
```

2.5.3　使用 neo4j-import 工具导入到 Neo4j

从 Neo4j 2.2 版本开始，系统就自带了一个大数据量的导入工具：neo4j-import，可支持并行、可扩展的大规模 CSV 数据导入。下面介绍使用这个工具导入 CSV 数据。

首先，可以在 Neo4j 系统目录下的 path/to/neo4j/bin/neo4j-import 路径下找到这个工具的可执行文件。下面是使用示例：

```
bin/neo4j-import --into retail.db --id-type string \
          --nodes:Customer customers.csv --nodes products.csv  \
          --nodes:orders_header.csv,orders1.csv,orders2.csv \
          --relationships:CONTAINS order_details.csv \
          --relationships:ORDERED
customer_orders_header.csv,orders1.csv,orders2.csv
```

上例中以“--nodes:”子句开头的 CSV 文件是节点 csv 文件；以“--relationships:”开头的是关系 CSV 文件；“--into”子句指明了导入的 Neo4j 数据库名称；“--id-type”子句指明了生成节点、关系的主键类型为 string 类型。

由于 neo4j-import 工具不能使用 Cypher 语句创建节点、关系，所以需要为节点和关系分别提供不同的 csv 文件。上例中各个文件的列的组织形式如表 2-3~表 2-7 所示。

表 2-3　customers.csv 文件格式

customerId:ID(Customer)	Name
23	Delicatessen Inc
42	Delicous Bakery

表 2-4　products.csv 文件格式

productId:ID(Product)	Name	price	:LABEL
11	Chocolate	10	Product;Food

表 2-5　orders_header.csv,orders1.csv,orders2.csv 文件格式

orderId:ID(Order)	date	total	customerId:IGNORE
1041	2015-05-10	130	23
1042	2015-05-12	20	42

表 2-6　order_details.csv 文件格式

:START_ID(Order)	amount	price	:END_ID(Product)
1041	13	130	11
1042	2	20	11

表 2-7　customer_orders_header.csv,orders1.csv,orders2.csv 文件格式

:END_ID(Order)	date:IGNORE	total:IGNORE	:START_ID(Customer)
1041	2015-05-10	130	23
1042	2015-05-12	20	42

对于上面各个 CSV 文件的内容解析如下：

（1）customers.csv 中的全部记录将作为节点直接导入到 Neo4j 数据库中，其中节点都带有 Customer 标签节点属性，直接从 CSV 文件的每条记录中获取。

（2）products.csv 文件导入形式与 customers.csv 相同，但其中节点标签取自:LABEL 列（Product 或者 Food）。

（3）对于节点的顺序取自 3 个文件（orders_header.csv、orders1.csv、orders2.csv），一个头文件和两个内容文件。

（4）接下来需要创建订单项（orders1.csv、orders2.csv）与所包含的产品（products.csv）相关联关系“CONTAINS”，这个关联关系是从 order_details.csv 创建的，通过其 ID 将订单与所包含的产品相关联。

（5）订单可以通过 orders1.csv、orders2.csv 文件关联到客户。

初学者可以参照本例中各个 CSV 文件的列格式制作自己的 CSV 文件，再使用 neo4j-import 工具进行导入。

第 3 章

◂ Neo4j 之 Cypher ▸

Cypher 是一种图数据库查询语言，表现力丰富，查询效率高，其地位和作用与关系型数据库中的 SQL 语言相当。Cypher 由 Neo Technology 公司为 Neo4j 而创建，公司为扩大 Cypher 影响力和使用范围，于 2015 年发布了 openCypher 工程将 Cypher 对外开放。

Cypher 语言一直在持续不断地发展和改进，不同版本之间存在语法上的细微差异。为充分展现 Cypher 的最新功能，本章以 Cypher 3.1 版本进行介绍。主要内容包括：

- Cypher 概述
- 基本语法
- Cypher 语句
- 函数
- 模式（Schema）
- 查询调优
- 执行计划

3.1 Cypher 概述

3.1.1 Cypher 是什么

Cypher 是一种声明式图数据库查询语言，它具有丰富的表现力，能高效地查询和更新图数据。对于初学者 Cypher 使用相对简单，但其功能还是非常强大，即便是非常复杂的数据库查询也能用 Cypher 简要地表达出来。这使得用户可以将精力集中在自己所从事的领域，而不用在数据库访问上花太多时间。

Cypher 查询语言设计很人性化，既适合开发人员，也适合专业的运营人员（这点尤为重要）。作为一个声明式查询语言，Cypher 专注于清晰地表达从图中检索什么，而不是怎么去检索。在这点上，与命令式的 Java 语言和脚本式的 Gremlin[4]语言完全不同 。

Cypher 博采众长，同时也继承了已有的惯用做法。像 WHERE 和 ORDER BY 等大多数

[4] http://gremlin.tinkerpop.com

关键词均来自于 SQL 语言[5]。而像模式匹配表达方法借鉴于 SPARQL 语言[6]，部分聚合（Collection）语法来源于像 Haskell 和 Python 语言。

Cypher 借鉴了 SQL 语言的结构——查询可由各种各样的语句组合。语句被链接在一起，相互之间传递中间结果集。查询语言由多种不同的语句构成，这里是一些获取图的常用语句：

- MATCH: 匹配图模式。这是从图中获取数据最常见的方法。
- WHERE: 不是独立的语句，而是 MATCH，OPTINAL MATCH 和 WITH 的一部分。用于给模式添加约束或者过滤传递给 WITH 的中间结果。
- RETURN: 定义返回的结果。

下面是 MATCH 和 RETURN 的例子。示例中的图数据如图 3-1 所示。

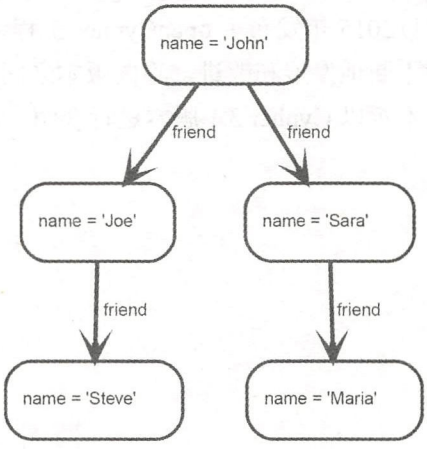


图 3-1 示例中的图数据

例如，下面是查找名为'John'和'John'朋友的朋友的查询语句。

```
MATCH (john {name: 'John'})-[:friend]->()-[:friend]->(fof)
RETURN john.name, fof.name
```

结果

```
+-----------+----------+
| john.name | fof.name |
+-----------+----------+
| "John"    | "Maria"  |
| "John"    | "Steve"  |
+-----------+----------+
2 rows
```



本章中这种框线输出格式是通过 neo4j-shell 得到的，有关 neo4j-shell 的使用请参见“第 2 章 Neo4j 基础入门”。

[5] http://en.wikipedia.org/wiki/SQL
[6] http://en.wikipedia.org/wiki/SPARQL

接下来在语句中添加一些过滤。

给定一个用户名列表，找到名字在列表中的所有节点。匹配他们的朋友，仅返回那些他们关注的 name 属性以'S'开头的用户。

```
MATCH (user)-[:friend]->(follower)
WHERE user.name IN ['Joe', 'John', 'Sara', 'Maria', 'Steve'] AND
follower.name =~ 'S.*'
RETURN user.name, follower.name
```

结果

```
+-----------+---------------+
| user.name | follower.name |
+-----------+---------------+
| "Joe"     | "Steve"       |
| "John"    | "Sara"        |
+-----------+---------------+
2 rows
```

下面是一些用于更新图常用的语句：

- CREATE（和 DELETE）：创建（和删除）节点和关系。
- SET（和 REMOVE）：使用 SET 设置属性值和给节点添加标签，使用 REMOVE 移除它们。
- MERGE：匹配已经存在的或者创建新节点和模式，这对于有唯一性约束的时候非常有用。

3.1.2　模式（Patterns）

Neo4j 图由节点和关系构成。节点可能还有标签和属性，关系可能还有类型和属性。节点表达的是实体，关系连接一对节点。节点可以看作类似关系数据库中的表，但又不完全一样。节点的标签可以理解为不同的表名，属性类似关系数据库中表的列。一个节点的数据类似关系数据库中表的一行数据。拥有相同标签的节点通常具有类似的属性，但不必完全一样，这点与关系数据库中一张表中的行数据拥有相同的列是不一样的。

然而，节点和关系都是简单的低层次的构建块。单个节点或者关系只能编码很少的信息，但模式可以将很多节点和关系编码为任意复杂的想法。

Cypher 查询语言很依赖于模式。只包含一个关系的简单模式连接了一对节点。例如，一个人 LIVES_IN 在某个城市或者某个城市 PART_OF 一个国家。使用了多个关系的复杂模式能够表达任意复杂的概念，可以支持各种有趣的使用场景。例如，下面的 Cypher 代码将两个简单的模式连接在一起：

```
(:Person) -[:LIVES_IN]-> (:City) -[:PART_OF]-> (:Country)
```

像关系数据库中的 SQL 一样，Cypher 是一种文本的声明式查询语言。它使用 ASCII art[7] 的形式来表达基于图的模式。采用类似 SQL 的语句，如 MATCH、WHERE 和 DELETE，来组合这些模式以表达所预期的操作。

[7] https://en.wikipedia.org/wiki/ASCII_art

3.1.2.1 节点语法

Cypher 采用一对圆括号来表示节点，如：()、(foo)。下面是一些常见的节点表示法：

```
()
(matrix)
(:Movie)
(matrix:Movie)
(matrix:Movie {title: "The Matrix"})
(matrix:Movie {title: "The Matrix", released: 1997})
```

简单的()表达了一个匿名节点。如果想在其他地方引用这个节点，可以添加一个变量，如(matrix)。此变量的可见范围局限于单个语句。

Movie 标签声明了节点的类型。Neo4j 节点索引也会使用到标签，每个索引都是建立在一个标签和属性的组合上。节点的属性以 key/value 列表的形式存在，并外加一对大括号。属性可以存储信息和（或者）限制模式。

3.1.2.2 关系语法

Cypher 使用一对短横线（即--）表示一个无方向关系。有方向的关系在其中一段加上一个箭头（即<--或-->）。方括号表达式[...]可用于添加详情。里面可以包含变量、属性和（或者）类型信息。关系的常见表达方式如下：

```
-->
-[role]->
-[:ACTED_IN]->
-[role:ACTED_IN]->
-[role:ACTED_IN {roles: ["Neo"]}]->
```

关系的方括号内的语法和语义与节点类似，定义了可以在别处引用的变量，关系的类型类似于节点的标签，关系的属性等同于节点的属性。注意，属性的值可以是数组。

3.1.2.3 模式语法

将节点和关系的语法组合在一起可以表达模式。下面是一个简单的模式：

```
(keanu:Person:Actor {name: "Keanu Reeves"})-[role:ACTED_IN {roles:
["Neo"]}]->
(matrix:Movie {title: "The Matrix"} )
```

3.1.2.4 模式变量

为了增强模块性和减少重复，Cypher 允许将模式赋给一个变量。这使得匹配到的路径可以用于其他表达式。如：

```
acted_in = (:Person)-[:ACTED_IN]->(:Movie)
```

3.1.3 查询和更新图

Cypher 语句既可用于查询，又可用于更新图数据。

3.1.3.1　更新语句的结构

一个 Cypher 查询部分不能同时匹配和更新图数据。每个部分要么读取和匹配图，要么更新它。

如果需要从图中读取，然后更新图，那么该查询隐含地包含两个部分——第一部分是读取，第二部分是写入。如果查询只是读取，Cypher 将采用惰性加载（Lazy Load），事实上并没匹配模式，直到需要返回结果时才实际地去匹配。在更新查询语句中，所有的读取操作必须在任何的写操作发生之前完成。

当希望使用聚合数据进行过滤时，必须使用 WITH 将两个读语句部分连接在一起。第一部分做聚合，第二部分过滤来自第一部分的结果。如下所示：

```
MATCH (n {name: 'John'})-[:FRIEND]-(friend)
WITH n, count(friend) AS friendsCount
WHERE friendsCount > 3
RETURN n, friendsCount
```

下面是一个将聚合数据更新到图中的例子：

```
MATCH (n {name: 'John'})-[:FRIEND]-(friend)
WITH n, count(friend) AS friendsCount
SET n.friendCount = friendsCount
RETURN n.friendsCount
```

在内存运行的情况下，可以尽可能多地将查询部分链接在一起。

3.1.3.2　返回数据

任何查询都可以返回数据。RETURN 语句有三个子语句，分别为 SKIP、LIMIT 和 ORDER BY。如果返回的图元素是刚刚删除的数据，需要注意的是这时数据的指针不再有效，针对它们的任何操作都是未定义的。

3.1.4　事务

任何更新图的查询都运行在一个事务中。一个更新查询要么全部成功，要么全部失败。Cypher 或者创建一个新的事务，或者运行在一个已有的事务中：

- 如果运行上下文中没有事务，Cypher 将创建一个，一旦查询完成就提交该事务。
- 如果运行上下文中已有事务，查询就会运行在该事务中。直到该事务成功地提交之后，数据才会持久化到磁盘中去。

可以将多个查询作为单个事务来提交：

（1）开始一个事务。
（2）运行多个 Cypher 更新查询。
（3）一次提交这些查询。



查询将这些变化放在内存中，直到整个查询执行完成。一个巨大的查询会导致 JVM 使用大量的堆空间。

3.1.5 唯一性

当进行模式匹配时，Neo4j 将确保单个模式中不会包含匹配到多次的同一个图关系。在大多数情况下，这是非常敏感的事。

例如：查找一个用户的朋友的朋友不应该返回该用户。

下面创建一些节点和关系：

```
CREATE (adam:User { name: 'Adam' }), (pernilla:User { name:'Pernilla' }),
(david:User { name: 'David'}), (adam)-[:FRIEND]->(pernilla),
(pernilla)-[:FRIEND]->(david)
```

上面的 Cypher 创建的图如图 3-2 所示。

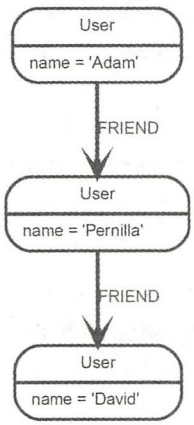


图 3-2 Cypher 创建的图

下面来查询 Adam 的朋友的朋友：

```
MATCH (user:User { name: 'Adam' })-[r1:FRIEND]-()-[r2:FRIEND]-
(friend_of_a_friend)
RETURN friend_of_a_friend.name AS fofName
```

结果

```
+-----------+
| fofName   |
+-----------+
| "David"   |
+-----------+
1 row
```

在这个查询中，Cypher 确保不会包含关系 r1 和 r2 指向的同一个图关系。

然后，有时也未必一直希望这样。如果查询应当返回该用户，可以通过多个 MATCH 语句延伸匹配关系来实现，如下：

```
MATCH (user:User { name: 'Adam' })-[r1:FRIEND]-(friend)
MATCH (friend)-[r2:FRIEND]-(friend_of_a_friend)
RETURN friend_of_a_friend.name AS fofName
```

结果

```
+---------+
| fofName |
+---------+
| "David" |
| "Adam"  |
+---------+
2 rows
```

注意，下面的查询看起来与前一个类似，但事实上它等价于再前一个。

```
MATCH (user:User { name: 'Adam' })-[r1:FRIEND]-(friend),(friend)-
[r2:FRIEND]-(friend_of_a_friend)
RETURN friend_of_a_friend.name AS fofName
```

这里的 MATCH 语句包含一个有两条路径的单个模式，而前一个查询有两个不同的模式。

```
+---------+
| fofName |
+---------+
| "David" |
+---------+
1 row
```

3.1.6　兼容性

Cypher 不是一成不变的语言。新版本引入了很多新的功能，一些旧的功能可能会被移除。如果需要，旧版本依然可以访问到。这里有两种方式在查询中选择使用哪个版本：

- 为所有查询设置版本：可以通过 neo4j.conf 中 cypher.default_language_version 参数来配置 Neo4j 数据库使用哪个版本的 Cypher 语言。
- 在查询中指定版本：简单地在查询开始的时候写上版本，如 Cypher 2.3。

下面是一个指定使用 Cypher 2.3 的查询例子：

```
CYPHER 2.3
START n=node:nodes(name = "A")
RETURN n
```

Neo4j 3.1 支持如下版本的 Cypher 语言：

- Neo4j Cypher 3.1
- Neo4j Cypher 3.0
- Neo4j Cypher 2.3

3.2 基本语法

本节主要描述 Cypher 查询语言的基本语法。

3.2.1 类型

Cypher 处理的所有值都有一个特定的类型，它支持如下类型：

- 数值型
- 字符串
- 布尔型
- 节点
- 关系
- 路径
- 映射（Map）
- 列表（List）

在 Cypher 语句中，大多数类型的值都可以使用字面值表达式 （参见“3.2.2 表达式”小节）。在使用 null 的时候要特别注意，因为 null 是任何类型的值（参见“3.2.9 空值”小节）。节点、关系和路径可以作为模式匹配的返回结果。



标签（label）不是值，它只是模式匹配的一种语法形式。

3.2.2 表达式

3.2.2.1 概述

Cypher 中的表达式如下：

- 十进制（整型和双精度型）的字面值：13、-4000、3.14、6.022E23。
- 十六进制整型字面值（以 0x 开头）：0x13zf、0xFC3A9、-0x66eff。
- 八进制整型字面值（以 0 开头）：01372、02127、 -05671。
- 字符串字面值：'Hello'、"World"。
- 布尔字面值：true、false、TRUE、FALSE。
- 变量：n、x、rel、myFancyVariable、`A name with weird stuff in it[]!`。
- 属性：n.prop、x.prop、rel.thisProperty 、myFancyVariable. `(weird property name)`。
- 动态属性：n["prop"]、rel[n.city + n.zip]、map[coll[0]]。
- 参数：$param、$0
- 表达式列表：['a', 'b']、[1, 2, 3]、['a', 2, n.property, $param] 、[ ]。

- 函数调用：length(p)、nodes(p)。
- 聚合函数：avg(x.prop)、count(*)。
- 路径-模式：(a)-->()<--(b)。
- 计算式：1 + 2 and 3 < 4。
- 返回 true 或者 false 的断言表达式：a.prop = 'Hello'、length(p) >10、exists(a.name)。
- 正则表达式：a.name =~ 'Tob.*'。
- 大小写敏感的字符串匹配表达式：a.surname STARTS WITH 'Sven'、a.surname ENDS WITH 'son' or a.surname CONTAINS 'son'。
- CASE 表达式。

3.2.2.2　转义字符

Cypher 中的字符串可以包含的转义字符如表 3-1 所示。

表 3-1　Cypher 中的字符串可以包含的转义字符

字符	含义
\t	制表符
\b	退格
\n	换行
\r	回车
\f	换页
\'	单引号
\"	双引号
\\	反斜杠
\uxxxx	Unicode UTF-16 编码点（4 位的十六进制数字必须以“\u”开头）
\uxxxxxxxx	Unicode UTF-32 编码点（8 位的十六进制数字必须以“\u”开头）

3.2.2.3　Case 表达式

Cypher 支持 Case 条件表达式，它类似于其他语言中的 if/else 语句。

1. 简单的 Case 表达式

计算表达式的值，然后依次与 WHEN 语句中的表达式进行比较，直到匹配上为止。如果未匹配上，则 ELSE 中的表达式将作为结果。如果 ELSE 语句不存在，那么将返回 null。

语法

```
CASE test
WHEN value THEN result
[WHEN ...]
[ELSE default]
END
```

参数

- test: 一个有效的表达式。
- value: 一个表达式，它的结果将与 test 表达式的结果进行比较。
- result: 如果 value 表达式能够与 test 表达式匹配，它将作为结果表达式。
- default: 没有匹配的情况下的默认返回表达式。

查询

```
MATCH (n)
RETURN
CASE n.eyes
WHEN 'blue'
THEN 1
WHEN 'brown'
THEN 2
ELSE 3 END AS result
```

结果

```
+--------+
| result |
+--------+
| 2      |
| 1      |
| 3      |
| 2      |
| 1      |
+--------+
5 rows
```

2. 一般的 Case 表达式

按顺序判断断言，直到找到一个为 true，然后对应的结果被返回。如果没有找到，就返回 ELSE 的值。如果没有 ELSE 语句，则返回 null。

语法

```
CASE
WHEN predicate THEN result
[WHEN ...]
[ELSE default]
END
```

参数

- predicate: 判断的断言，以找到一个有效的可选项。
- result: 如果 predicate 匹配到，result 就作为结果表达式。
- default: 没有匹配到情况下，默认返回表达式。

查询

```
MATCH (n)
RETURN
```

```
CASE
WHEN n.eyes = 'blue'
THEN 1
WHEN n.age < 40
THEN 2
ELSE 3 END AS result
```

结果

```
+--------+
| result |
+--------+
| 2      |
| 1      |
| 3      |
| 3      |
| 1      |
+--------+
5 rows
```

3.2.3　变量

当需要引用模式（Pattern）或者查询某一部分的时候，可以对其进行命名。针对不同部分的这些命名被称为变量。

例如：

```
MATCH (n)-->(b)
RETURN b
```

这里的 n 和 b 就是变量。

变量名是区分大小写的。它可以包含下划线、字母（a~z、A~Z）和数字（0~9），但必须以字母开头。如果变量名中需要用到其他字符，可以用反向单引号（`）将变量名括起来。比如，如果变量名中包含特殊字符 μ，则可以按如下方式使用：

```
match(`μg`:food) return `μg`
```

变量的命名规则同样也适用于属性的命名。



变量仅在同一个查询内可见。它不能被用于后续的查询。如果有 WITH 连接起来的多个查询部分，变量必须列在 WITH 语句中才能应用到后续部分。详细参见“3.3.18 WITH”小节。

3.2.4　参数

Cypher 支持带参数的查询，这意味着开发人员不是必须用字符串来构建查询。此外，这也让执行计划的缓存更容易。

参数能够用于 WHERE 语句中的字面值和表达式，START 语句中的索引值、索引查询以及节点和关系的 id。参数不能用于属性名、关系类型和标签，因为这些模式（Pattern）将作

为查询结构的一部分被编译进查询计划。

合法的参数名是字母、数字以及两者的组合。下面是一个使用参数的完整例子。参数以JSON 格式提供，具体如何提交它们取决于所使用的驱动程序。

3.2.4.1 字符串

参数

```
{
"name" : "Johan"
}
```

可以有两种查询方式来使用参数，如下：

查询方式 1

```
MATCH (n)
WHERE n.name = $name
RETURN n
```

查询方式 2

```
MATCH (n { name: $name })
RETURN n
```

3.2.4.2 正则表达式

参数

```
{
"regex" : ".*h.*"
}
```

查询

```
MATCH (n)
WHERE n.name =~ $regex
RETURN n.name
```

3.2.4.3 大小写敏感的字符串模式匹配

参数

```
{
"name" : "Michael"
}
```

查询

```
MATCH (n)
WHERE n.name STARTS WITH $name
RETURN n.name
```

3.2.4.4　创建带有属性的节点

参数

```
{
"props" : {
"name" : "Andres",
"position" : "Developer"
}
}
```

查询

```
CREATE ($props)
```

3.2.4.5　创建带有多个属性的多个节点

参数

```
{
"props" : [ {
"awesome" : true,
"name" : "Andres",
"position" : "Developer"
}, {
"children" : 3,
"name" : "Michael",
"position" : "Developer"
} ]
}
```

查询

```
UNWIND $props AS properties
CREATE (n:Person)
SET n = properties
RETURN n
```

3.2.4.6　设置节点的所有属性

注意：这将替换当前的所有属性。

参数

```
{
"props" : {
"name" : "Andres",
"position" : "Developer"
}
}
```

查询

```
MATCH (n)
WHERE n.name='Michaela'
SET n = $props
```

3.2.4.7 SKIP 和 LIMIT

参数

```
{
"s" : 1,
"l" : 1
}
```

查询

```
MATCH (n)
RETURN n.name
SKIP $s
LIMIT $l
```

3.2.4.8 节点 id

参数

```
{
"id" : 0
}
```

查询

```
MATCH (n)
WHERE id(n)= $id
RETURN n.name
```

3.2.4.9 多个节点 id

参数

```
{
"ids" : [ 0, 1, 2 ]
}
```

查询

```
MATCH (n)
WHERE id(n) IN $ids
RETURN n.name
```

3.2.5　运算符

3.2.5.1　数学运算符

数学运算符包括+、-、*、/、%和^。

3.2.5.2　比较运算符

比较运算符包括=、<>、<、>、<=、>=、IS NULL 和 IS NOT NULL。

3.2.5.3　布尔运算符

布尔运算符包括 AND、OR、XOR 和 NOT。

3.2.5.4　字符串运算符

连接字符串的运算符为+，正则表达式的匹配运算符为=~。

3.2.5.5　列表运算符

列表的连接也可以通过+运算符，可以用 IN 来检查列表中是否存在某个元素。

3.2.5.6　属性运算符



Cypher 2.0 版本以后，之前存在的属性运算符“?”和“!”已经被移除了。这个语法不再支持。对于不存在的属性将返回 null。如果真的还需要?运算符的功能，可以使用(NOT(has(<ident>.prop)) OR <ident>.prop=<value>)。使用“? ”表达可选关系也被移除了，取而代之的是 OPTINAL MATCH。

3.2.5.7　值的相等与比较

Cypher 支持使用=和<>来比较两个值的相等/不相等关系，同类型的值只有它们是同一个值的时候才相等，如 3 = 3 和"x" <> "xy"。

只有两个 Map 的键相同且指向的值也相等的时候它们才相等。对于列表来说，只有它们包含相等值的相同序列的时候才相等，如[3, 4] = [1+2, 8/2]。

不同类型的值在比较相等的时候遵循以下规则：

- 路径可看作是一些节点和关系的列表，它等于所有包含相同序列节点和关系的所有列表。
- 对任何值测试是否 = 和 <> null 都将返回 null，这包括 null = null 和 null <> null。唯一可靠地测试一个值是否为 null 的方法是使用 IS NULL 或者 IS NOT NULL。

不同类型之间不能相互比较。特别地，节点、关系和映射之间不能相互比较。

3.2.5.8 值的排序与比较

比较运算符<=、<（升序）和>=、>（降序）可以用于值排序的比较，如下所示。

- 数字型值的排序比较采用数字顺序。
- java.lang.Double.NaN 大于所有值。
- 字符串排序的比较采用字典顺序。如"x" < "xy"。
- 布尔值的排序遵循 false < true。
- 当有个参数为 null 的时候，比较结果为 null，如 null < 3 的结果为 null。
- 将其他类型的值相互比较进行排序将报错。

3.2.5.9 链式比较运算

比较运算可以被任意地联结在一起，如 x < y <= z 等价于 x < y AND y <= z。

正式地，如果 a, b, c, …, y, z 是表达式，op1, op2, …, opN 是比较运算符，这时，a op1 b op2 c … y opN z 等价于 a op1 b and b op2 c and … y opN z。

例如：

```
MATCH (n) WHERE 21 < n.age <= 30 RETURN n
```

等价于：

```
MATCH (n) WHERE 21 < n.age AND n.age <= 30 RETURN n
```

该查询将匹配年龄介于 21 和 30 之间的所有节点。

3.2.6 注释

Cypher 语言的注释类似其他语言，用双斜线//来注释行，例如：

```
MATCH (n) RETURN n //这是行末尾注释
MATCH (n)
//这是整行注释
RETURN n
MATCH (n) WHERE n.property = '//这不是注释' RETURN n
```

3.2.7 模式（Patterns）

模式和模式匹配是 Cypher 非常核心的部分。要高效地使用 Cypher 必须深入理解模式。

使用模式可以描述你期望看到的数据的形状。例如，在 MATCH 语句中，当用模式描述一个形状的时候，Cypher 将按照模式来获取相应的数据。

模式描述数据的形式很类似在白板上画出图的形状。通常用圆圈来表达节点，使用箭头来表达关系。

模式在 MATCH、CREATE 和 MERGE 等语句中都会出现，后续章节会详细描述。

3.2.7.1　节点模式

模式能表达的最简单的形状就是节点。节点使用一对圆括号表示，然后中间含一个名字。例如：

```
(a)
```

这个模式描述了一个节点，其名称使用变量 a 表示。

3.2.7.2　关联节点的模式

模式可以描述多个节点及其之间的关系。Cypher 使用箭头来表达两个节点之间的关系。例如：

```
(a)-->(b)
```

这个模式描述了一个非常简单的数据形状，即两个节点和从其中一个节点到另外一个节点的关系。两个节点分别命名为 a 和 b，关系是有方向的，从 a 指向 b。

这种描述节点和关系的方式可以扩展到任意数量的节点和它们之间的关系，例如：

```
(a)-->(b)<--(c)
```

这一系列相互关联的节点和关系被称为路径（Path）。

注意：节点的命名仅仅当后续的模式或者 Cypher 查询中需要引用时才需要。如果不需要引用，则命名可以省略。例如：

```
(a)-->()<--(c)
```

3.2.7.3　标签

模式除了可以描述节点之外，还可以用来描述标签。比如：

```
(a:User)-->(b)
```

也可以描述一个节点的多个标签，如：

```
(a:User:Admin)-->(b)
```

3.2.7.4　指定属性

节点和关系是图的基础结构。Neo4j 的节点和关系都可以有属性，这样可以建立更丰富的模型。属性在模式中使用键值对的映射结构来表达，然后用大括号包起来。例如，一个有两个属性的节点如下所示。

```
(a {name: 'Andres', sport: 'Brazilian Ju-Jitsu'})
```

关系中的属性如下所示。

```
(a)-[{blocked: false}]->(b)
```

当模式中有属性时，它实际上为数据增加了额外的约束。在 CREATE 语句中，属性会被增加到新创建的节点和关系中。在 MERGE 语句中，属性将作为一个约束去匹配数据库中的

仅供非商业用途或交流学习使用

数据是否存在该属性。如果没有匹配到，这时 MERGE 的行为将与 CREATE 一样，即属性将被设置到新创建的节点和关系中。



模式在 CREATE 语句中支持使用单个参数来指定属性。例如：CREATE (node $paramName)。但这在其他语句中是不行的，因为 Cypher 在编译查询的时候需要知道属性的名称，以便能够高效地匹配。

3.2.7.5 描述关系

如前面的例子所示，可以用箭头简单地描述两个节点之间的关系。它描述了关系的存在性和方向性。但如果不关心关系的方向，则箭头的头部可以省略。例如：

```
(a)--(b)
```

与节点类似，如果后续需要引用到该关系，则可以给关系赋一个变量名。变量名需要用方括号括起来，放在箭头的短横线中间，如下所示。

```
(a)-[r]->(b)
```

就像节点有标签一样，关系可以有类型（Type）。给关系指定类型，如下所示。

```
(a)-[r:REL_TYPE]->(b)
```

不像节点可以有多个标签，关系只能有一个类型。但如果所描述的关系可以是一个类型集中的任意一种类型，可以将这些类型都列入模式中，它们之间以竖线“|”分割，例如：

```
(a)-[r:TYPE1|TYPE2]->(b)
```

注意：这种模式仅适用于描述已经存在的数据（如在 MATCH 语句中），而在 CREATE 或者 MERGE 语句中是不允许的，因为一个关系不能创建多个类型。

与节点类似，关系的命名也是可以省略的，例如：

```
(a)-[:REL_TYPE]->(b)
```

与使用一串节点和关系来描述一个长路径的模式不同，很多关系（以及中间的节点）可以采用指定关系的长度的模式来描述，例如：

```
(a)-[*2]->(b)
```

它描述了一张三个节点和两个关系的图。这些节点和关系都在同一条路径中（路径的长度为 2）。它等同于：

```
(a)-->()-->(b)
```

关系的长度也可以指定一个范围，这被称为可变长度的关系，例如：

```
(a)-[*3..5]->(b)
```

关系的长度最小值为 3，最大值为 5。它描述了一个或者有 4 个节点和 3 个关系，或者 5 个节点 4 个关系，或者 6 个节点和 5 个关系连在一起的图组成的一条路径。

仅供非商业用途或交流学习使用

长度的边界也是可以省略的，如描述一个路径长度大于等于 3 的路径：

```
(a)-[*3..]->(b)
```

路径长度小于等于 5 的路径，例如：

```
(a)-[*..5]->(b)
```

两个边界都可以省略，这允许任意长度的路径，例如：

```
(a)-[*]->(b)
```

来看一个简单的查询例子：

查询

```
MATCH (me)-[:KNOWS*1..2]-(remote_friend)
WHERE me.name = 'Filipa'
RETURN remote_friend.name
```

结果

```
+--------------------+
| remote_friend.name |
+--------------------+
| "Dilshad"          |
| "Anders"           |
+--------------------+
2 rows
```

这个查询用于找到符合这个模式的数据：即指定一个节点（name 属性值为‘Filipa’）和与它关系为 KNOWS 的一步和两步的节点。这是一个查询一度和二度人脉的典型例子。



变长关系不能用于 CREATE 和 MERGE 语句。

3.2.7.6　赋值给路径变量

如上所述，连接在一起的一系列节点和关系被称为路径。Cypher 允许使用标识符给路径命名，例如：

```
p= (a)-[*3..5]->(b)
```

在 MATCH 中，CREAT 和 MERGE 语句中可以这样做，但当模式作为表达式的时候不能这样。

3.2.8　列表

Cypher 对列表（List）有很好的支持。

3.2.8.1　概述

可以使用方括号和一组以逗号分割的元素来创建一个列表。

仅供非商业用途或交流学习使用

查询

```
RETURN [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] AS list
```

结果

```
+-----------------------+
| list                  |
+-----------------------+
| [0,1,2,3,4,5,6,7,8,9] |
+-----------------------+
1 row
```

在下面的例子中使用了 range 函数。它指定了列表包含元素的开始数字和结束数字。范围的两端也是被包含在内的，例如：

查询

```
RETURN range(0, 10)
```

结果

```
+--------------------------+
| list                     |
+--------------------------+
| [0,1,2,3,4,5,6,7,8,9,10] |
+--------------------------+
1 row
```

可以使用方括号[]访问列表中的元素。

查询

```
RETURN range(0, 10)[3]
```

结果

```
+-----------------+
| range(0, 10)[3] |
+-----------------+
| 3               |
+-----------------+
1 row
```

索引也可以为负数，这时访问的方向将从列表的末尾作为开始点。

查询

```
RETURN range(0, 10)[-3]
```

结果

```
+------------------+
| range(0, 10)[-3] |
+------------------+
| 8                |
+------------------+
1 row
```


仅供非商业用途或交流学习使用

也可以在[]中指定列表返回指定范围的元素。它将提取开始索引到结束索引的值，但不包含结束索引所对应的值。如下面例子中，开始索引为 0，结束索引为 3，结果 0,1,2 索引对应的值被返回，但结束索引 3 对应的值未返回。

查询

```
RETURN range(0, 10)[0..3]
```

结果

```
+--------------------+
| range(0, 10)[0..3] |
+--------------------+
| [0,1,2]            |
+--------------------+
1 row
```

查询

```
RETURN range(0, 10)[0..-5]
```

结果

```
+----------------------+
| range(0, 10)[0..-5] |
+----------------------+
| [0,1,2,3,4,5]        |
+----------------------+
1 row
```

查询

```
RETURN range(0, 10)[-5..]
```

结果

```
+--------------------+
| range(0, 10)[-5..] |
+--------------------+
| [6,7,8,9,10]       |
+--------------------+
1 row
```

查询

```
RETURN range(0, 10)[..4]
```

结果

```
+-------------------+
| range(0, 10)[..4] |
+-------------------+
| [0,1,2,3]         |
+-------------------+
1 row
```



如果返回一个范围的索引值越界了，那么返回直接从越界的地方进行截断。如果是单个元素的索引值越界，则返回 null。

仅供非商业用途或交流学习使用

查询

```
RETURN range(0, 10)[5..15]
```

结果

```
+---------------------+
| range(0, 10)[5..15] |
+---------------------+
| [5,6,7,8,9,10]      |
+---------------------+
1 row
```

查询

```
RETURN range(0, 10)[15]
```

结果

```
+-------------------+
| range(0, 10)[15] |
+-------------------+
| <null>           |
+-------------------+
1 row
```

可以用 size 函数获取列表的长度，例如：

查询

```
RETURN size(range(0, 10)[0..3])
```

结果

```
+--------------------------+
| size(range(0, 10)[0..3]) |
+--------------------------+
| 3                        |
+--------------------------+
1 row
```

3.2.8.2 List 推导式

List 推导式（Comprehension）是 Cypher 中基于已经存在的列表创建一个列表的语法构造。它遵循数学上的集合，代替使用映射和过滤函数。

查询

```
RETURN [x IN range(0,10) WHERE x % 2 = 0 | x^3] AS result
```

结果

```
+----------------------------------+
| result                           |
+----------------------------------+
| [0.0,8.0,64.0,216.0,512.0,1000.0] |
+----------------------------------+
1 row
```


仅供非商业用途或交流学习使用

如果希望分别地过滤或者映射，WHERE 部分或者表达式部分都是可以省略的。

查询

```
RETURN [x IN range(0,10) WHERE x % 2 = 0] AS result
```

结果

```
+----------------+
| result         |
+----------------+
| [0,2,4,6,8,10] |
+----------------+
1 row
```

查询

```
RETURN [x IN range(0,10)| x^3] AS result
```

结果

```
+------------------------------------------------------------------+
| result                                                           |
+------------------------------------------------------------------+
| [0.0,1.0,8.0,27.0,64.0,125.0,216.0,343.0,512.0,729.0,1000.0] |
+------------------------------------------------------------------+
1 row
```

3.2.8.3　模式推导式

模式推导式是 Cypher 基于模式匹配的结果创建列表的一种语法构造。模式推导式将像一般的 MATCH 语句那样去匹配模式，断言部分与一般的 WHERE 语句一样，但它将产生一个指定的定制投射。

查询

```
MATCH (a:Person { name: 'Charlie Sheen' })
RETURN [(a)-->(b)
WHERE b:Movie | b.year] AS years
```

结果

```
+------------------+
| years            |
+------------------+
| [1979,1984,1987] |
+------------------+
1 row
```

整个断言，包括 WHERE 关键字都是可选的，可以被省略。

3.2.8.4　字面值映射

Cypher 也可以构造映射。通过 REST 接口可以获取 JSON 对象。在 Java 中对应的是 java.util.Map<String, Object>。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

查询

```
RETURN { key: 'Value', listKey: [{ inner: 'Map1' }, { inner: 'Map2' }]}
```

结果

```
+-------------------------------------------------------------------+
| {key: 'Value', listKey: [{inner: 'Map1'}, {inner: 'Map2'}]}       |
+-------------------------------------------------------------------+
| {key -> "Value", listKey -> [{inner -> "Map1"},{inner -> "Map2"}]} |
+-------------------------------------------------------------------+
1 row
```

3.2.8.5 Map 投射

Cypher 支持一个名为“map projections”的概念。它使得基于已有的节点、关系和其他 map 值来构建 map 变得容易。

Map 投射以指向图实体的且用逗号分隔的变量簇开头，并包含以{}包括起来的映射元素，语法如下：

```
map_variable {map_element, [, …n]}
```

一个 map 元素投射一个或多个键值对到 map 投射。这里有 4 种类型的 map 投射元素：

- 属性选择器——投射属性名作为键，map_variable 中对应键的值作为键值。
- 字面值项——来自任意表达式的键值对，如 key: <expression>。
- 变量选择器——投射一个变量，变量名作为键，变量的值作为投射的值。它的语法只有变量。
- 全属性选择器——投射来自 map_variable 中的所有键值对。



如果 map_variable 的值指向一个 null，那么整个 map 投射将返回 null。

投射举例

找到 Charlie Sheen 和返回关于他和他参演过的电影。这个例子展示了字面值项类型的 map 投射，反过来它还在聚合函数 collect()中使用了 map 投射。

查询

```
MATCH (actor:Person { name: 'Charlie Sheen' })-[:ACTED_IN]->(movie:Movie)
RETURN actor { .name, .realName, movies: collect(movie { .title, .year })}
```

结果

```
+-------------------------------------------------------------------------------+
| actor                                                                         |
+-------------------------------------------------------------------------------+
| {name -> "Charlie Sheen", realName -> "Carlos Irwin Estevez", movies ->       |
|  [{title -> "Apocalypse Now",year -> 1979},{title -> "Red Dawn", year ->      |
|  1984},{title -> "Wall Street", year -> 1987}]}                               |
+-------------------------------------------------------------------------------+
1 row
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

找到演过电影的所有演员，并显示他们所参演电影的数量。这个例子用一个变量来代表数量，使用变量选择器来投射值，如下所示。

查询

```
MATCH (actor:Person)-[:ACTED_IN]->(movie:Movie)
WITH actor, count(movie) AS nrOfMovies
RETURN actor { .name, nrOfMovies }
```

结果

```
+------------------------------------------+
| actor                                    |
+------------------------------------------+
| {name -> "Charlie Sheen", nrOfMovies -> 3} |
| {name -> "Martin Sheen", nrOfMovies -> 2}  |
+------------------------------------------+
2 rows
```

还是以“Charlie Sheen”为例，此时返回该节点的所有属性。这里使用了全属性选择器来投射所有的节点属性和一个额外的显式投射的 age 属性。因为此属性在该节点不存在，所以投射的值为 null。

查询

```
MATCH (actor:Person { name: 'Charlie Sheen' })
RETURN actor { .*, .age }
```

结果

```
+-------------------------------------------------------------------------------+
| actor                                                                         |
+-------------------------------------------------------------------------------+
| {realName -> "Carlos Irwin Estevez", name -> "Charlie Sheen", age -> <null>} |
+-------------------------------------------------------------------------------+
1 row
```

3.2.9　空值

3.2.9.1　空值介绍

空值 null 在 Cypher 中表示未找到或者未定义。从概念上讲，null 意味着“一个未找到的未知值”。对待 null 会与其他值有些不同，例如从节点中获取一个并不存在的属性将返回 null。大多数以 null 作为输入的表达式将返回 null。这包括 WHERE 语句中用于断言的布尔表达式。

null 不等于 null，两个未知的值并不意味着它们是同一个值。因此，null = null 返回 null 而不是 true。

3.2.9.2　空值的逻辑运算

逻辑运算符包括 AND、OR、XOR、IN、NOT，把 null 当作未知的三值逻辑值，AND、OR 和 XOR 的逻辑值表如表 3-2 所示。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

表 3-2　空值与 AND、OR 和 XOR 的逻辑值表

a	b	a AND b	a OR b	a XOR b
false	false	false	false	false
false	null	false	null	null
false	true	false	true	true
true	false	false	true	true
true	null	null	true	null
true	true	true	true	false
null	false	false	null	null
null	null	null	null	null
null	true	null	true	null

3.2.9.3　空值与 IN

IN 运算符遵循类似的逻辑。如果列表中存在某个值，结果就返回 true；如果列表包含 null 值并且没有匹配到值，结果返回 null；否则结果为 false。表 3-3 给出一些例子。

表 3-3　空值与 IN 运算符的例子

表达式	结果
2 IN [1, 2, 3]	true
2 IN [1, null, 3]	null
2 IN [1, 2, null]	true
2 IN [1]	false
2 IN []	false
null IN [1, 2, 3]	null
null IN [1, null, 3]	null
null IN []	false

All、any、none 和 single 与 IN 类似，如果可以确切地计算结果，将返回 true 或者 false；否则将返回 null。

3.2.9.4　返回空值的表达式

- 从列表中获取不存在的元素：[][0]，head([])。
- 试图访问节点或者关系的不存在的属性：n.missingProperty。
- 与 null 做比较：1 < null。
- 包含 null 的算术运算：1 + null。
- 包含任何 null 参数的函数调用：sin(null)。

仅供非商业用途或交流学习使用

3.3 语句

语句可分为三类，包括读语句、写语句和通用语句。

- 读语句：MATCH、OPTIONAL MATCH、WHERE、START、Aggregation 和 LOAD CSV。
- 写语句：CREATE、MERGE、SET、DELETE、REMOVE、FOREACH 和 CREATE UNIQUE。
- 通用语句：RETURN、ORDER BY、LIMIT、SKIP、WITH、UNWIND、UNION 和 CALL。

3.3.1 MATCH

MATCH 语句用指定的模式检索数据库。

3.3.1.1 简介

MATCH 语句通过模式（Pattern）来检索数据库。它常与带有约束或者断言的 WHERE 语句一起使用，这使得匹配的模式更具体。断言是模式描述的一部分，不能看作是匹配结果的过滤器。这意味着 WHERE 应当总是与 MATCH 语句放在一起使用。

MATCH 可以出现在查询的开始或者末尾，也可能位于 WITH 之后。如果它在语句开头，此时不会绑定任何数据。Neo4j 将设计一个搜索去找到匹配这个语句以及 WHERE 中指定断言的结果。这将牵涉数据库的扫描，搜索特定标签的节点或者搜索一个索引以找到匹配模式的开始点。这个搜索找到的节点和关系可作为一个“绑定模式元素（Bound Pattern Elements）”。它可以用于匹配一些子图的模式，也可以用于任何进一步的 MATCH 语句，Neo4j 将使用这些已知的元素来找到更进一步的未知元素。

Cypher 是声明式的，因此查询本身不指定搜索的算法。Neo4j 会自动地用最好的方法去找到开始节点和匹配模式。WHERE 中的断言可以在模式匹配之前、匹配中或者匹配后进行处理。这可以通过查询编译器来影响这个决定。详细可参见“3.5.1 索引”小节和“3.6.4 USING”小节。

下面的例子基于如图 3-3 所示的图例。

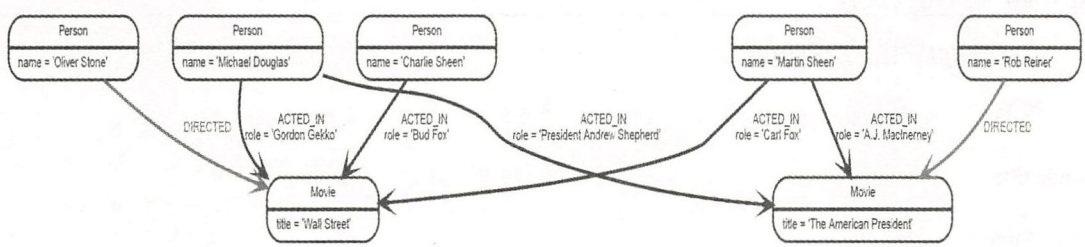


图 3-3　MATCH 语句示例

仅供非商业用途或交流学习使用

3.3.1.2 查找节点

1. 查询所有节点

通过指定一个不带标签的节点的模式，图中的所有节点将返回。

查询

```
MATCH (n)
RETURN n
```

返回数据库中的所有节点。

结果

N
Node[0]\{name:"Charlie Sheen"\}
Node[1]\{name:"Martin Sheen"\}
Node[2]\{name:"Michael Douglas"\}
Node[3]\{name:"Oliver Stone"\}
Node[4]\{name:"Rob Reiner"\}
Node[5]\{title:"Wall Street"\}
Node[6]\{title:"The American President"\}
7 rows

2. 查询带有某个标签的所有节点

通过指定带有一个标签的节点的模式，可以获取满足该标签的所有节点。

查询

```
MATCH (movie:Movie)
RETURN movie.title
```

返回数据库中的所有电影。

结果

movie.title
"Wall Street"
"The American President"

仅供非商业用途或交流学习使用

2 rows

查询关联节点

符号--意为相关的，这个关系不带有类型和方向。

查询

```
MATCH (director { name: 'Oliver Stone' })--(movie)
RETURN movie.title
```

返回'Oliver Stone'导演的所有电影。

结果

movie.title
"Wall Street"
1 row

3. 匹配标签

可以为查询的节点增加标签约束。

查询

```
MATCH (:Person { name: 'Oliver Stone' })--(movie:Movie)
RETURN movie.title
```

返回与 Person 'Oliver'相连的带有 Movie 标签的所有节点。

结果

movie.title
"Wall Street"
1 row

3.3.1.3　查找关系

1. 外向关系

关系的方向通过-->或者<--来表示。

查询

```
MATCH (:Person { name: 'Oliver Stone' })-->(movie)
RETURN movie.title
```

返回与 Person 'Oliver Stone'外向连接的所有节点。

仅供非商业用途或交流学习使用

结果

movie.title
"Wall Street"
1 row

2. 有向关系和变量

当需要过滤关系中的属性，或者返回关系的时候，变量就很有必要了。

查询

```
MATCH (:Person { name: 'Oliver Stone' })-[r]->(movie)
RETURN type(r)
```

返回'Oliver'的外向关系的类型。

结果

type(r)
"DIRECTED"
1 row

3. 匹配关系类型

当已知要匹配关系的类型时，可通过冒号后面紧跟关系类型。

查询

```
MATCH (wallstreet:Movie { title: 'Wall Street' })<-[:ACTED_IN]-(actor)
RETURN actor.name
```

返回'Wall Street'中的所有演员。

结果

actor.name
"Michael Douglas"
"Martin Sheen"
"Charlie Sheen"
3 rows

仅供非商业用途或交流学习使用

4. 匹配多种关系类型

当需要匹配多种关系中的一种时，可以通过竖线（|）将多个关系连接在一起。

查询

```
MATCH (wallstreet { title: 'Wall Street' })<-[:ACTED_IN|:DIRECTED]-(person)
RETURN person.name
```

返回与'Wall Street'节点关系为 ACTED_IN 或者 DIRECTED 的所有节点。

结果

person.name
"Oliver Stone"
"Michael Douglas"
"Martin Sheen"
"Charlie Sheen"
4 rows

5. 匹配关系类型和使用关系变量

如果想通过变量来引用关系和指定关系类型，可以将它们放在一起。例如：

查询

```
MATCH (wallstreet { title: 'Wall Street' })<-[r:ACTED_IN]-(actor)
RETURN r.role
```

返回'Wall Street'中所有演员的角色。

结果

r.role
"Gordon Gekko"
"Carl Fox"
"Bud Fox"
3 rows

3.3.1.4　关系的深度

1. 带有特殊字符的关系类型

某些时候数据库中可能会有非字母字符的类型，或者中间含有空格。可以使用反引号

“`”将它们括起来。下面的例子在'Charlie Sheen'和'Rob Reiner'之间添加了一个关系。该关系中包含有空格。

查询

```
MATCH (charlie:Person { name: 'Charlie Sheen' }),(rob:Person { name: 'Rob
Reiner' })
CREATE (rob)-[:`TYPE WITH SPACE`]->(charlie)
```

该查询产生的结果如图 3-4 所示。

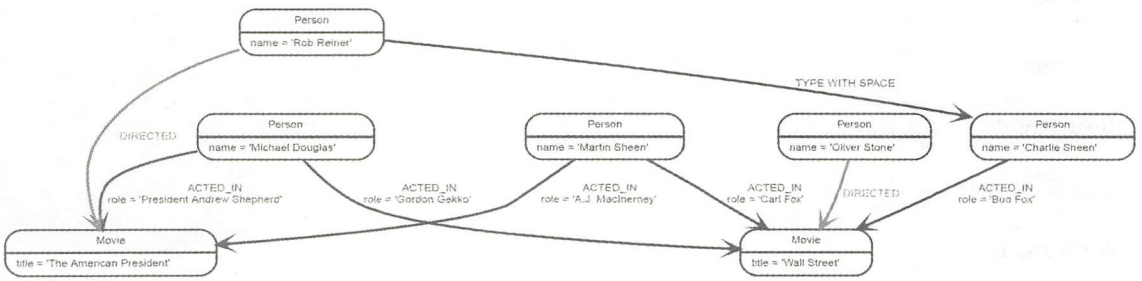


图 3-4　查询结果图

查询

```
MATCH (n { name: 'Rob Reiner' })-[r:`TYPE WITH SPACE`]->()
RETURN type(r)
```

返回带有空格的关系类型。

结果

type(r)
"TYPE WITH SPACE"
type(r)
1 row

2. 多个关系

关系可以多语句以 ()--()的形式来表达，或者它们相互连接在一起。例如：

查询

```
MATCH (charlie { name: 'Charlie Sheen' })-[:ACTED_IN]->(movie)<-[:DIRECTED]-
(director)
RETURN movie.title, director.name
```

返回'Charlie Sheen'演的电影和该电影的导演。

结果

movie.title	director.name
"Wall Street"	"Oliver Stone"
1 row	

3. 可变长关系

可变长关系和节点的语法如下：

```
-[:TYPE*minHops..maxHops]->
```

minHops 和 maxHops 都是可选的，默认值分别为 1 和无穷大。当没有边界值的时候，点也可以省略。当只设置了一个边界的时候，如果点省略了就意味着是一个固定长度的模式。

查询

```
MATCH (martin { name: 'Charlie Sheen' })-[:ACTED_IN*1..3]-(movie:Movie)
RETURN movie.title
```

返回与'Charlie Sheen'关系为 1 跳（Hop）到 3 跳的所有电影。

结果

movie.title
"Wall Street"
"The American President"
"The American President"
3 rows

4. 变长关系的关系变量

当连接两个节点之间的长度是变长的，那么关系变量返回的将可能是一个关系列表。

查询

```
MATCH (actor { name: 'Charlie Sheen' })-[r:ACTED_IN*2]-(co_actor)
RETURN r
```

返回一个关系列表。

结果

R
[:ACTED_IN[0]\{role:"Bud Fox"\},:ACTED_IN[1]\{role:"Carl Fox"\}]
[:ACTED_IN[0]\{role:"Bud Fox"\},:ACTED_IN[2]\{role:"Gordon Gekko"\}]
2 rows



5. 变长路径上的属性匹配

带有属性的可变长度关系意味着路径上的所有关系都必须包含给定的属性值。在这个查询中，'Charlie Sheen'和他的父亲'Martin Sheen'之间有两条路径。其中一条包含一个'blocked'关系，另外一条则没有。首先，通过下面的语句增加 BLOCKED 和 UNBLOCKED 关系。

查询

```
MATCH (charlie:Person { name: 'Charlie Sheen' }),(martin:Person { name:
'Martin Sheen' })
CREATE (charlie)-[:X { blocked: FALSE }]->(:UNBLOCKED)<-[:X { blocked:
FALSE }]-(martin)
CREATE (charlie)-[:X { blocked: TRUE }]->(:BLOCKED)<-[:X { blocked:
FALSE }]-(martin)
```

后续将以图 3-5 为基础来讲解。

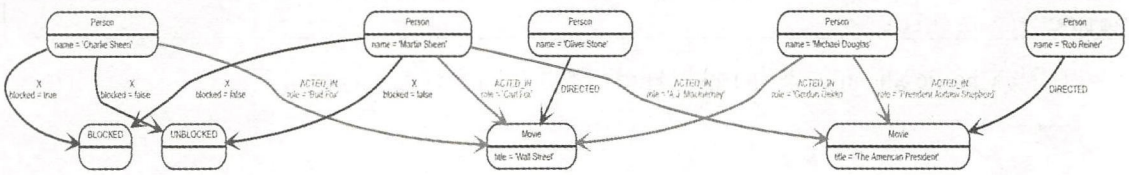


图 3-5　变长路径上的属性匹配图例

查询

```
MATCH p =(charlie:Person)-[* { blocked:false }]-(martin:Person)
WHERE charlie.name = 'Charlie Sheen' AND martin.name = 'Martin Sheen'
RETURN p
```

返回'Charlie Sheen'和'Martin Sheen'之间满足 blocked 属性值为 false 的所有关系。

结果

P
[Node[0]\{name:"Charlie Sheen"\},:X[7]\{blocked:false\},Node[7]\{\},:X[8]\{blocked:false\},Node[1]\{name:"Martin Sheen"\}]
1 row

6. 零长路径

如果变长路径的下界值为零，则意味着两个变量指向同一个节点。

查询

```
MATCH (wallstreet:Movie { title: 'Wall Street' })-[*0..1]-(x)
RETURN x
```

返回电影本身及一跳关系的演员和导演。

结果

X
Node[5]\{title:"Wall Street"\}
Node[0]\{name:"Charlie Sheen"\}
Node[1]\{name:"Martin Sheen"\}
Node[2]\{name:"Michael Douglas"\}
Node[3]\{name:"Oliver Stone"\}
5 rows

7. 命名路径

如果想返回或者需要对路径进行过滤，可以将路径赋值给一个变量。

查询

```
MATCH p =(michael { name: 'Michael Douglas' })-->()
RETURN p
```

返回从开始'Michael Douglas'的两条路径。

结果

P
[Node[2]\{name:"Michael Douglas"\},:ACTED_IN[5]\{role:"President Andrew Shepherd"\},Node[6]\{title:"The American President"\}]
[Node[2]\{name:"Michael Douglas"\},:ACTED_IN[2]\{role:"Gordon Gekko"\},Node[5]\{title:"Wall Street"\}]
2 rows

8. 匹配一簇关系

当模式包含一簇关系时，关系模式不会指定方向，Cypher 将尝试匹配两个方向的关系。

查询

```
MATCH (a)-[r]-(b)
WHERE id(r)= 0
RETURN a,b
```

返回两个相连的节点，一个为开始节点，一个为结束节点。



结果

A	b
Node[0]\{name:"Charlie Sheen"\}	Node[5]\{title:"Wall Street"\}
Node[5]\{title:"Wall Street"\}	Node[0]\{name:"Charlie Sheen"\}
2 rows	

3.3.1.5 最短路径

1. 单条最短路径

通过使用 shortestPath 函数很容易找到两个节点之间的最短路径，如下所示。

查询

```
MATCH (martin:Person { name: 'Martin Sheen' }),(oliver:Person { name:
'Oliver Stone' }), p =shortestPath((martin)-[*..15]-(oliver))
RETURN p
```

上面查询的含义为：找到两个节点之间的最短路径，路径最大长度为 15。在搜索最短路径的时候，还可以使用关系类型、最大跳数和方向等约束条件。如果用到了 WHERE 语句，相关的断言会被包含到 shortestPath 中去。如果路径的关系元素中用到了 none() 或者 all() 断言，那么这些将用于在检索时提高性能。

结果

P
[Node[1]\{name:"Martin Sheen"\},:ACTED_IN[1]\{role:"Carl Fox"\},Node[5]\{title:"Wall Street"\},:DIRECTED[3]\{\},Node[3]\{name:"Oliver Stone"\}]
1 row

2. 带断言的最短路径

最短路径模式中 WHERE 语句的断言通常在检索匹配的最短路径之前就进行处理了。

查询

```
MATCH (charlie:Person { name: 'Charlie Sheen' }),(martin:Person { name:
'Martin Sheen' }), p =shortestPath((charlie)-[*]-(martin))
WHERE NONE (r IN rels(p) WHERE type(r)= 'FATHER')
RETURN p
```

这个查询寻找'Charlie Sheen'和'Martin Sheen'之间的最短路径，WHERE 断言确保不考虑两个节点之间的父亲/儿子关系。



结果

P
[Node[0]\{name:"Charlie Sheen"\},:ACTED_IN[0]\{role:"Bud Fox"\},Node[5]\{title:"Wall Street"\},:ACTED_IN[1]\{role:"Carl Fox"\},Node[1]\{name:"Martin Sheen"\}]
1 row

3. 所有最短路径

找到两个节点之间的所有最短路径。

查询

```
MATCH (martin:Person { name: 'Martin Sheen' }),(michael:Person { name:
'Michael Douglas' }), p = allShortestPaths((martin)-[*]-(michael))
RETURN p
```

找到了'Martin Sheen'和'Michael Douglas'之间的两条最短路径。

结果

P
[Node[1]\{name:"Martin Sheen"\},:ACTED_IN[1]\{role:"Carl Fox"\},Node[5]\{title:"Wall Street"\},:ACTED_IN[2]\{role:"Gordon Gekko"\},Node[2]\{name:"Michael Douglas"\}]
[Node[1]\{name:"Martin Sheen"\},:ACTED_IN[4]\{role:"A.J. MacInerney"\},Node[6]\{title:"The American President"\},:ACTED_IN[5]\{role:"President Andrew Shepherd"\},Node[2]\{name:"Michael Douglas"\}]
2 rows

3.3.1.6　通过 id 查询节点或关系

1. 通过 id 查询节点

可以在断言中使用 id()函数来根据 id 查询节点。



Neo4j 会重用已删除的节点和关系的内部 id。这意味着依赖 Neo4j 内部的 id 存在风险。因此，建议通过程序来产生 id。

查询

```
MATCH (n)
WHERE id(n)= 0
RETURN n
```

节点 id 为 0 的节点将返回。



结果

N
Node[0]\{name:"Charlie Sheen"\}
1 row

2. 通过 id 查询关系

通过 id 查询关系与节点类似，但这在实践中不推荐这么做。

查询

```
MATCH ()-[r]->()
WHERE id(r)= 0
RETURN r
```

关系 id 为 0 的关系将返回。

结果

R
:ACTED_IN[0]\{role:"Bud Fox"\}
1 row

3. 通过 id 查询多个节点

通过 id 查询多个节点的时候，可以将 id 放到 IN 语句中。

查询

```
MATCH (n)
WHERE id(n) IN [0, 3, 5]
RETURN n
```

这个查询返回了 IN 语句中列出的所有节点。

结果

N
Node[0]\{name:"Charlie Sheen"\}
Node[3]\{name:"Oliver Stone"\}
Node[5]\{title:"Wall Street"\}
3 rows



3.3.2　OPTINAL MATCH

OPTINAL MATCH 语句用于搜索模式中描述的匹配项，对于找不到的项用 null 代替。

3.3.2.1　简介

OPTINAL MATCH 匹配模式与 MATCH 类似。不同之处在于，如果没有匹配到，OPTINAL MATCH 将用 null 作为未匹配到部分的值。OPTINAL MATCH 在 Cypher 中类似 SQL 语句中的 outer join。

要么匹配整个模式，要么都未匹配。记住，WHERE 是模式描述的一部分，匹配的时候就会考虑到 WHERE 语句中的断言，而不是匹配之后。这对于有多个(OPTINAL)MATCH 语句的查询尤其重要，一定要将属于 MATCH 的 WHERE 语句与 MATCH 放在一起。

OPTINAL MATCH 图例如图 3-6 所示。

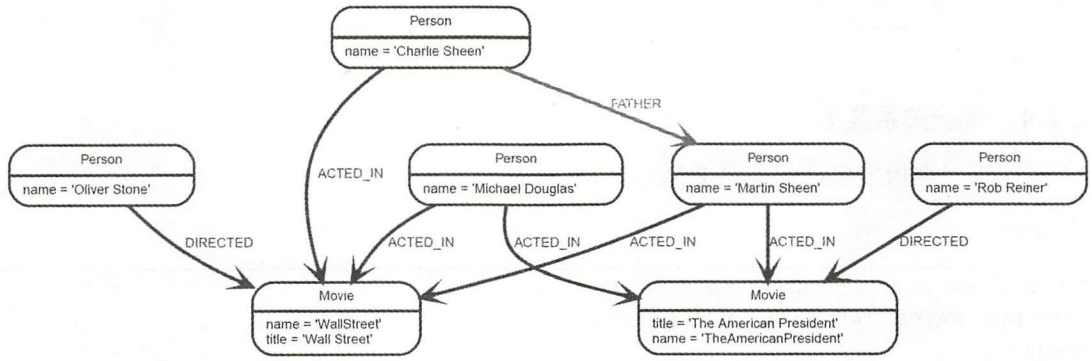


图 3-6　OPTINAL MATCH 图例

3.3.2.2　关系

如果某个关系是可选的，可使用 OPTINAL MATCH。这很类似 SQL 中 outer join 的工作方式。如果关系存在就返回，否则在相应的地方返回 null。

查询

```
MATCH (a:Movie { title: 'Wall Street' })
OPTIONAL MATCH (a)-->(x)
RETURN x
```

返回了 null，因为这个节点没有外向关系。

结果

```
+--------+
| x      |
+--------+
| <null> |
+--------+
1 row
```



3.3.2.3 可选元素的属性

如果可选的元素为 null，那么该元素的属性也返回 null。

查询

```
MATCH (a:Movie { title: 'Wall Street' })
OPTIONAL MATCH (a)-->(x)
RETURN x, x.name
```

返回了 x 元素（查询中为 null），它的 name 属性也为 null。

结果

```
+--------+--------+
| x      | x.name |
+--------+--------+
| <null> | <null> |
+--------+--------+
1 row
```

3.3.2.4 可选关系类型

可在查询中指定可选的关系类型。

查询

```
MATCH (a:Movie { title: 'Wall Street' })
OPTIONAL MATCH (a)-[r:ACTED_IN]->()
RETURN r
```

返回了 null 关系，因为该节点没有 ACTS_IN 的外向关系。

结果

```
+--------+
| r      |
+--------+
| <null> |
+--------+
1 row
```

3.3.3 WHERE

WHERE 在 MATCH 或者 OPTINAL MATCH 语句中添加约束，或者与 WITH 一起使用来过滤结果。

WHERE 不能单独使用，它只能作为 MATCH、OPTINAL MATCH、START 和 WITH 的一部分。如果是在 WITH 和 START 中，它用于过滤结果。对于 MATCH 和 OPTINAL MATCH，WHERE 为模式增加约束，它不能看作是匹配完成后的结果过滤。

WHERE 图例如图 3-7 所示。

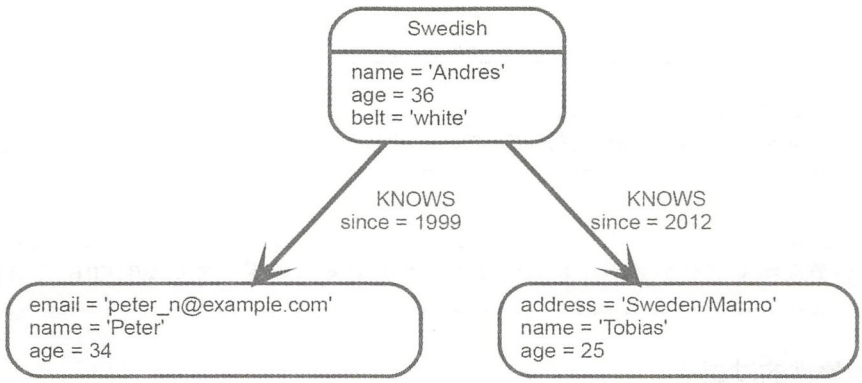


图 3-7　WHERE 图例

3.3.3.1　基本使用

1. 布尔运算

可以在 WHERE 中使用布尔运算符，如 AND、OR，以及布尔函数 NOT。

查询

```
MATCH (n)
WHERE n.name = 'Peter' XOR (n.age < 30 AND n.name = 'Tobias') OR NOT (n.name
= 'Tobias' OR n.name ='Peter')
RETURN n
```

结果

```
+--------------------------------------------------------------+
| n                                                            |
+--------------------------------------------------------------+
| Node[0]{name:"Andres",age:36,belt:"white"}                   |
| Node[1]{address:"Sweden/Malmo",name:"Tobias",age:25}         |
| Node[2]{email:"peter_n@example.com",name:"Peter",age:34}     |
+--------------------------------------------------------------+
3 rows
```

2. 节点标签的过滤

可以在 WHERE 中类似使用 WHERE n:foo 写入标签断言来过滤节点。

查询

```
MATCH (n)
WHERE n:Swedish
RETURN n
```

将返回“Andres”节点。

结果

```
+-------------------------------------------+
| n                                         |
+-------------------------------------------+
| Node[0]{name:"Andres",age:36,belt:"white"} |
+-------------------------------------------+
1 row
```



如果要查询不包含某个标签的其他所有节点的反向查询，可在 WHERE 后加 NOT。

3. 节点属性的过滤

可以在 WHERE 语句中对节点的属性进行过滤。

查询

```
MATCH (n)
WHERE n.age < 30
RETURN n
```

返回了“Tobias”节点，因为他的年龄小于 30。

结果

```
+-----------------------------------------------------+
| n                                                   |
+-----------------------------------------------------+
| Node[1]{address:"Sweden/Malmo",name:"Tobias",age:25} |
+-----------------------------------------------------+
1 row
```

4. 关系属性的过滤

要对关系的属性进行过滤，可在 WHERE 中添加如下关键词：

查询

```
MATCH (n)-[k:KNOWS]->(f)
WHERE k.since < 2000
RETURN f
```

返回了“Peter”，因为 Andres 自 1999 年就认识他了。

结果

```
+-------------------------------------------------------------+
| f                                                           |
+-------------------------------------------------------------+
| Node[2]{email:"peter_n@example.com",name:"Peter",age:34} |
+-------------------------------------------------------------+
1 row
```

5. 动态节点属性过滤

以方括号语法的形式可使用动态计算的值来过滤属性。

参数

```
{
"prop" : "AGE"
}
```

查询

```
MATCH (n)
WHERE n[toLower($prop)]< 30
RETURN n
```

返回了“Tobias”，因为他的年龄小于 30。

结果

```
+-----------------------------------------------------+
| n                                                   |
+-----------------------------------------------------+
| Node[1]{address:"Sweden/Malmo",name:"Tobias",age:25} |
+-----------------------------------------------------+
1 row
```

6. 属性存在性检查

使用 exists()只能检查节点或者关系的某个属性是否存在。

查询

```
MATCH (n)
WHERE exists(n.belt)
RETURN n
```

“Andres”返回了，因为只有他有 belt 属性。



has()函数已被移除，并被 exists()替代了。

结果

```
+----------------------------------------+
| n                                      |
+----------------------------------------+
| Node[0]{name:"Andres",age:36,belt:"white"} |
+----------------------------------------+
1 row
```

3.3.3.2　字符串匹配

可以用 START WITH 和 ENDS WITH 来匹配字符串的开始和结尾。如果不关心所匹配字符串的位置，可以用 CONTAINS，匹配是区分大小写的。

1. 匹配字符串的开始

STARTS WITH 用于以大小写敏感的方式匹配字符串的开始。

查询

```
MATCH (n)
WHERE n.name STARTS WITH 'Pet'
RETURN n
```

“Peter”返回了，因为他的名字以“Pet”开始。

结果

```
+-------------------------------------------------------+
| n                                                     |
+-------------------------------------------------------+
| Node[2]{email:"peter_n@example.com",name:"Peter",age:34} |
+-------------------------------------------------------+
1 row
```

2. 匹配字符串的结尾

ENDS WITH 用于以大小写敏感的方式匹配字符串的结尾。

查询

```
MATCH (n)
WHERE n.name ENDS WITH 'ter'
RETURN n
```

“Peter”返回了，因为他的名字以“ter”结尾。

结果

```
+-------------------------------------------------------+
| n                                                     |
+-------------------------------------------------------+
| Node[2]{email:"peter_n@example.com",name:"Peter",age:34} |
+-------------------------------------------------------+
1 row
```

3. 字符串包含

CONTAINS 用于检查字符串中是否包含某个字符串，它是大小写敏感的，且不关心匹配部分在字符串中的位置。

查询

```
MATCH (n)
WHERE n.name CONTAINS 'ete'
RETURN n
```

“Peter”返回了，因为他的名字包含“ete”字符串。

结果

```
+-------------------------------------------------------+
| n                                                     |
+-------------------------------------------------------+
| Node[2]{email:"peter_n@example.com",name:"Peter",age:34} |
```

```
+-------------------------------------------------------+
1 row
```

4. 字符串反向匹配

使用 NOT 关键词可以返回不满足给定字符串匹配要求的结果。

查询

```
MATCH (n)
WHERE NOT n.name ENDS WITH 's'
RETURN n
```

“Peter”返回了，因为他的名字不以“s”结尾。

结果

```
+-------------------------------------------------------------+
| n                                                           |
+-------------------------------------------------------------+
| Node[2]{email:"peter_n@example.com",name:"Peter",age:34} |
+-------------------------------------------------------------+
1 row
```

3.3.3.3　正则表达式

Cypher 支持正则表达式过滤。正则表达式的语法继承来自 Java 正则表达式[8]。它支持字符串如何匹配标记，包括不区分大小写（?i），多行（?m）和单行（?s）。标记放在正则表达式的开头，例如 MATCH (n) WHERE n.name =~ '(?i)Lon.*' RETURN n 将返回名字为'London'和'LonDoN'的节点。

1. 正则表达式

可以使用=~ 'regexp'来进行正则表达式的匹配。

查询

```
MATCH (n)
WHERE n.name =~ 'Tob.*'
RETURN n
```

“Tobias”返回了，因为他的名字以'Tob'开始。

结果

```
+-------------------------------------------------------+
| n                                                     |
+-------------------------------------------------------+
| Node[1]{address:"Sweden/Malmo",name:"Tobias",age:25} |
+-------------------------------------------------------+
1 row
```

[8] https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

2. 正则表达式中的转义字符

如果需要在正则表达式中插入斜杠，需使用转义字符。注意：字符串中的反斜杠也需要转义。

查询

```
MATCH (n)
WHERE n.address =~ 'Sweden\\/Malmo'
RETURN n
```

“Tobias”返回了，因为他的地址在“Sweden/Malmo”。

结果

```
+-------------------------------------------------------+
| n                                                     |
+-------------------------------------------------------+
| Node[1]{address:"Sweden/Malmo",name:"Tobias",age:25} |
+-------------------------------------------------------+
1 row
```

3. 正则表达式的非大小写敏感

在正则表达式前面加入(?i)之后，整个正则表达式将变成非大小写敏感。

查询

```
MATCH (n)
WHERE n.name =~ '(?i)ANDR.*'
RETURN n
```

“Andres”返回了，因为他的名字在不考虑大小写的情况下以“ANDR”开始。

结果

```
+-----------------------------------------+
| n                                       |
+-----------------------------------------+
| Node[0]{name:"Andres",age:36,belt:"white"} |
+-----------------------------------------+
1 row
```

3.3.3.4 在 WHERE 中使用路径模式

1. 模式过滤

模式是返回一个路径列表的表达式。列表表达式也是断言，空列表代表 false，非空列表代表 true。因此，模式不仅仅是表达式，同时也是断言。模式的局限性在于只能在单条路径中表达它，不能像在 MATCH 语句中那样使用逗号分隔多条路径，但可以通过 AND 组合多个模式。



不能在 WHERE 中的模式引入新的变量。尽管它看起来与 MATCH 中的模式类似。但 MATCH (a)-[]→(b)与 WHERE (a)-[]→(b)有很大的不同，前者将产生一个它匹配到的 a 和 b 之间路径子图，而后者是排除匹配到的 a 和 b 之间没有一个有向关系链的任何子图。

查询

```
MATCH (tobias { name: 'Tobias' }),(others)
WHERE others.name IN ['Andres', 'Peter'] AND (tobias)<--(others)
RETURN others
```

返回了有外向关系指向“Tobias”的节点。

结果

```
+-------------------------------------------+
| others                                    |
+-------------------------------------------+
| Node[0]{name:"Andres",age:36,belt:"white"} |
+-------------------------------------------+
1 row
```

2. 模式中的 NOT 过滤

NOT 功能可用于排除某个模式。

查询

```
MATCH (persons),(peter { name: 'Peter' })
WHERE NOT (persons)-->(peter)
RETURN persons
```

返回没有外向关系指向‘Peter’的节点。

结果

```
+-----------------------------------------------------------+
| persons                                                   |
+-----------------------------------------------------------+
| Node[1]{address:"Sweden/Malmo",name:"Tobias",age:25}      |
| Node[2]{email:"peter_n@example.com",name:"Peter",age:34} |
+-----------------------------------------------------------+
2 rows
```

3. 模式中的属性过滤

可以在模式中添加属性来过滤结果。

查询

```
MATCH (n)
WHERE (n)-[:KNOWS]-({ name: 'Tobias' })
RETURN n
```

返回与节点'Tobias'有 KNOWS 关系的所有节点。

结果

```
+-----------------------------------------+
| n                                       |
+-----------------------------------------+
| Node[0]{name:"Andres",age:36,belt:"white"} |
+-----------------------------------------+
1 row
```

4. 关系类型过滤

可以在 MATCH 模式中添加关系类型，但有时候希望在类型过滤上具有丰富的功能。这时，可以将类型与其他进行比较。例如，下面是一个对关系类型与一个正则表达式进行比较的例子。

查询

```
MATCH (n)-[r]->()
WHERE n.name='Andres' AND type(r)=~ 'K.*'
RETURN r
```

这个查询返回与'Andres'节点以‘K’开始的所有关系。

结果

```
+-----------------------+
| r                     |
+-----------------------+
| :KNOWS[1]{since:1999} |
| :KNOWS[0]{since:2012} |
+-----------------------+
2 rows
```

3.3.3.5 列表

IN 运算符

检查列表中是否存在某个元素，可以使用 IN 运算符。

查询

```
MATCH (a)
WHERE a.name IN ['Peter', 'Tobias']
RETURN a
```

这个查询检查字符串列表中是否存在某个属性。

结果

```
+-------------------------------------------------------------+
| a                                                           |
+-------------------------------------------------------------+
| Node[1]{address:"Sweden/Malmo",name:"Tobias",age:25}        |
| Node[2]{email:"peter_n@example.com",name:"Peter",age:34}    |
+-------------------------------------------------------------+
2 rows
```

3.3.3.6　不存在的属性和值

如果属性不存在，对它的判断默认返回 false。

对于不存在的属性值就当作 null，在下面例子中，对于没有 belt 属性的节点的比较将返回 false。

查询

```
MATCH (n)
WHERE n.belt = 'white'
RETURN n
```

仅返回了 belt 为 white 的节点。

结果

```
+-------------------------------------------+
| n                                         |
+-------------------------------------------+
| Node[0]{name:"Andres",age:36,belt:"white"} |
+-------------------------------------------+
1 row
```

1. 属性不存在默认为 true 的情况

如果要比较的属性存在，则可以与期望的值进行比较。如果不存在（IS NULL），默认值为 true。如：

查询

```
MATCH (n)
WHERE n.belt = 'white' OR n.belt IS NULL RETURN n
ORDER BY n.name
```

返回了满足 belt 属性值为 white 和不存在 belt 属性的所有节点。

结果

```
+-------------------------------------------------------------+
| n                                                           |
+-------------------------------------------------------------+
| Node[0]{name:"Andres",age:36,belt:"white"}                  |
| Node[2]{email:"peter_n@example.com",name:"Peter",age:34}    |
| Node[1]{address:"Sweden/Malmo",name:"Tobias",age:25}        |
+-------------------------------------------------------------+
3 rows
```

2. 空值过滤

有时候需要测试某个值或变量是否为 null。在 Cypher 中与 SQL 类似，可以使用 IS NULL。相反，“不为空”使用 IS NOT NULL，尽管 NOT (IS NULL x)也可以。

查询

```
MATCH (person)
```

```
WHERE person.name = 'Peter' AND person.belt IS NULL RETURN person
```

返回 name 属性值为'Peter'的且不存在 belt 属性的节点。

结果

```
+-----------------------------------------------------------+
| person                                                    |
+-----------------------------------------------------------+
| Node[2]{email:"peter_n@example.com",name:"Peter",age:34} |
+-----------------------------------------------------------+
1 row
```

3.3.3.7 使用范围

1. 简单范围

检查某个元素是否在指定的范围，可以使用不等运算符<，>=和>。

查询

```
MATCH (a)
WHERE a.name >= 'Peter'
RETURN a
```

返回节点的 name 属性的字典顺序大于或等于'Peter'的节点。

结果

```
+-----------------------------------------------------------+
| a                                                         |
+-----------------------------------------------------------+
| Node[1]{address:"Sweden/Malmo",name:"Tobias",age:25}      |
| Node[2]{email:"peter_n@example.com",name:"Peter",age:34} |
+-----------------------------------------------------------+
2 rows
```

2. 范围的组合

多个不等式可以组合构成一个范围。

查询

```
MATCH (a)
WHERE a.name > 'Andres' AND a.name < 'Tobias'
RETURN a
```

返回 name 属性值的字典顺序介于'Andres'和'Tobias'之间的节点。

结果

```
+-----------------------------------------------------------+
| a                                                         |
+-----------------------------------------------------------+
| Node[2]{email:"peter_n@example.com",name:"Peter",age:34} |
+-----------------------------------------------------------+
1 row
```

3.3.4　START

通过遗留索引（Legacy Index）查找开始点。



START 语句应当仅用于访问遗留的索引。所有其他的情况，都应使用 MATCH 代替。

Cypher 中的每个查询描述了一个模式，一个模式可以有多个开始点。一个开始点是模式中的一个关系或者节点。使用 START 时，只能通过遗留索引寻找来引出开始点。注意，使用一个不存在的遗留索引将报错。

START 图例如图 3-8 所示。

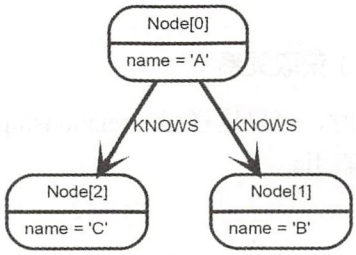


图 3-8　START 图例

3.3.4.1　通过索引获取节点

1. 通过索引搜索（Index Seek）获取节点

当采用索引搜索查找开始点时，可以这样做：node:index-name(key = "value")。在本例中存在一个名为 nodes 的节点索引。

查询

```
START n = node:nodes(name = 'A')
RETURN n
```

结果

```
+-------------------+
| n                 |
+-------------------+
| Node[0]{name:"A"} |
+-------------------+
1 row
```

2. 通过索引查询（Index Query）获取节点

当采用复杂的 Lucene 查询来查找开始点时，语法为：node:index-name("query")。这样可以写出很高级的索引查询出来。

查询

```
START n = node:nodes("name:A")
RETURN n
```

结果

```
+--------------------+
| n                  |
+--------------------+
| Node[0]{name:"A"} |
+--------------------+
1 row
```

3.3.4.2 通过索引获取关系

通过索引搜索（Index Seek）获取关系

当采用索引搜索查找开始点时，可以这样做：relationship:index-name(key = "value")。在本例中存在一个名为 rels 的关系索引。

查询

```
START r = relationship:rels(name = 'Andres')
RETURN r
```

结果

```
+--------------------------+
| r                        |
+--------------------------+
| :KNOWS[0]{name:"Andres"} |
+--------------------------+
1 row
```

3.3.5 Aggregation

3.3.5.1 简介

Cypher 支持使用聚合（Aggregation）来计算聚在一起的数据，类似 SQL 中的 group by。聚合函数有多个输入值，然后基于它们计算出一个聚合值。例如，avg 函数计算多个数值的平均值。min 函数用于找到一组值中最小的那个值。

聚合可以在匹配到的子图上进行计算。非聚合的表达式将值聚集起来，然后放入聚合函数。

例如，下面的返回语句：

```
RETURN n, count(*)
```

这里有两个表达式：n 和 count()。前者 n 不是聚合函数，是一个分组键。后者 count()是一个聚合函数。因此，根据不同的分组键（Grouping Key），匹配的子图将被分为不同的组。聚合函数将运行在这些组上来计算聚合值。

下面的例子对理解这个很有帮助，该查询基于如图 3-9 所示的数据。

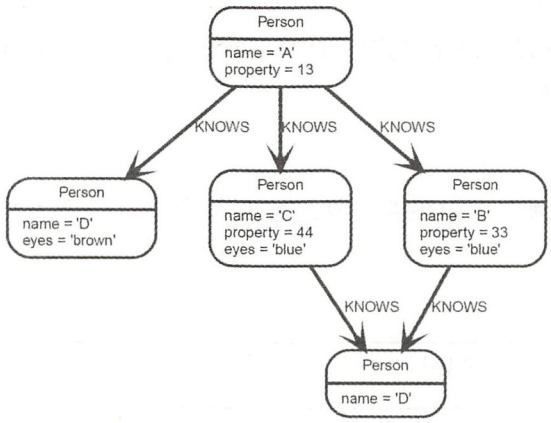


图 3-9　查询例子的数据

查询

```
MATCH (me:Person)-->(friend:Person)-->(friend_of_friend:Person)
WHERE me.name = 'A'
RETURN count(DISTINCT friend_of_friend), count(friend_of_friend)
```

在这个例子中，试图找到朋友的所有朋友并计算朋友的个数。第一个 count(DISTINCT friend_of_friend)聚合函数中每个 friend_of_friend 只会计算一次，因为 DISTINCT 剔除了重复的部分。第二个聚合函数中，每个 friend_of_friend 会被计算多次。当没有使用 DISTINCT 时，因为 B 和 C 都认识 D，因此 D 被计算了两次。

结果

```
+----------------------------------+--------------------------+
| count(distinct friend_of_friend) | count(friend_of_friend) |
+----------------------------------+--------------------------+
| 1                                | 2                        |
+----------------------------------+--------------------------+
1 row
```

聚合图例如图 3-10 所示。

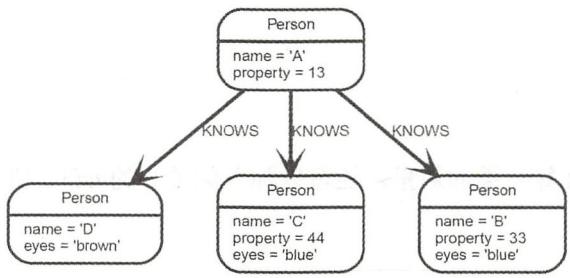


图 3-10　聚合图例

3.3.5.2 Count

count 用于计算行的数量。

count 有两种使用方式：count(*)用于计算匹配的行数，而 count(<expression>)用于计算<expression>中非空值的数量。

1. 计算节点

计算节点的数量。例如：如果要计算连接到某个节点的节点数，可用 count(*)如下所示。

查询

```
MATCH (n { name: 'A' })-->(x)
RETURN n, count(*)
```

返回了开始节点及与之相连节点的数量。

结果

```
+-------------------------------+----------+
| n                             | count(*) |
+-------------------------------+----------+
| Node[0]{name:"A",property:13} | 3        |
+-------------------------------+----------+
1 row
```

2. 按组计算关系类型的数量

计算关系类型组中的数量，返回类型和数量。

查询

```
MATCH (n { name: 'A' })-[r]->()
RETURN type(r), count(*)
```

返回关系类型和关系组中的关系数量。

结果

```
+---------+----------+
| type(r) | count(*) |
+---------+----------+
| "KNOWS" | 3        |
+---------+----------+
1 row
```

3. 计算实体

除了通过 count(*)计算结果的数量，还可以加入你关心的 name 值。

查询

```
MATCH (n { name: 'A' })-->(x)
RETURN count(x)
```

返回与满足带有 name 属性值为'A'的节点相连的所有节点的数量。

结果

```
+----------+
| count(x) |
+----------+
| 3        |
+----------+
1 row
```

4. 计算非空值的数量

可以通过 count(expression)来计算非空值的数量。

查询

```
MATCH (n:Person)
RETURN count(n.property)
```

返回了 property 属性非空的所有节点。

结果

```
+-------------------+
| count(n.property) |
+-------------------+
| 3                 |
+-------------------+
1 row
```

3.3.5.3　统计

1. sum

聚合函数 sum 简单地计算所有值之和。计算的时候，空值将被丢弃。

查询

```
MATCH (n:Person)
RETURN sum(n.property)
```

返回包含 Person 标签的所有节点的 property 属性值的和。

结果

```
+-----------------+
| sum(n.property) |
+-----------------+
| 90              |
+-----------------+
1 row
```

2. avg

avg 计算数值列的平均值。

查询

```
MATCH (n:Person)
RETURN avg(n.property)
```

返回 property 属性值的平均值。

结果

```
+-----------------+
| avg(n.property) |
+-----------------+
| 30.0            |
+-----------------+
1 row
```

3. percentileDisc

percentileDisc 计算给定值在一个组中的百分位，取值从 0.0 到 1.0。它使用舍入法，返回最接近百分位的值。对于插值法，请参考 percentileCont 函数。

查询

```
MATCH (n:Person)
RETURN percentileDisc(n.property, 0.5)
```

结果

```
+---------------------------------+
| percentileDisc(n.property, 0.5) |
+---------------------------------+
| 33                              |
+---------------------------------+
1 row
```

4. percentileCont

percentileCont 计算给定值在一个组中的百分位，百分位的值从 0.0 到 1.0。它采用线性插值的方法，在两个值之间计算一个加权平均数。对于使用舍入法获取最近的值，请参考 percentileDisc 函数。

查询

```
MATCH (n:Person)
RETURN percentileCont(n.property, 0.4)
```

结果

```
+---------------------------------+
| percentileCont(n.property, 0.4) |
+---------------------------------+
| 29.0                            |
+---------------------------------+
1 row
```

5. stdev

stdev 计算给定值在一个组中的标准偏差。它采用标准的 two-pass 方法，以 N-1 作为分

母。当以部分样本作为无偏估计时应使用 stdev；当计算整个样本的标准偏差时，应该使用 stdevp。

查询

```
MATCH (n)
WHERE n.name IN ['A', 'B', 'C']
RETURN stdev(n.property)
```

结果

```
+--------------------+
| stdev(n.property)  |
+--------------------+
| 15.716233645501712 |
+--------------------+
1 row
```

6. stdevp

stdevp 计算给定值在一个组中的标准偏差。与 stdev 类似，区别如上所述。

查询

```
MATCH (n)
WHERE n.name IN ['A', 'B', 'C']
RETURN stdevp(n.property)
```

结果

```
+--------------------+
| stdevp(n.property) |
+--------------------+
| 12.832251036613439 |
+--------------------+
1 row
```

max

max 查找数值列中的最大值。

查询

```
MATCH (n:Person)
RETURN max(n.property)
```

返回了 property 属性中的最大值。

结果

```
+-----------------+
| max(n.property) |
+-----------------+
| 44              |
+-----------------+
1 row
```

7. min

min 查找数值列中的最小值。

查询

```
MATCH (n:Person)
RETURN min(n.property)
```

返回了 property 属性中的最小值。

结果

```
+-----------------+
| min(n.property) |
+-----------------+
| 13              |
+-----------------+
1 row
```

3.3.5.4 Collect

collect 将所有的值收集起来放入一个列表，空值 null 将被忽略。

查询

```
MATCH (n:Person)
RETURN collect(n.property)
```

以列表的形式返回收集到的值。

结果

```
+---------------------+
| collect(n.property) |
+---------------------+
| [13,33,44]          |
+---------------------+
1 row
```

3.3.5.5 DISTINCT

所有的聚合函数都可以带有 DISTINCT 修饰符，它将去掉其中的重复值。因此，计算节点中不重复眼睛颜色数量的查询可以这样写：

查询

```
MATCH (a:Person { name: 'A' })-->(b)
RETURN count(DISTINCT b.eyes)
```

结果

```
+------------------------+
| count(DISTINCT b.eyes) |
+------------------------+
| 2                      |
+------------------------+
1 row
```

3.3.6　LOAD CSV

LOAD CSV 用于从 CSV 文件中导入数据。

- CSV 文件的 URL 可以由 FROM 后面紧跟的任意表达式来指定。
- 需要使用 AS 来为 CSV 数据指定一个变量。
- LOAD CSV 支持以 gzip、Deflate 和 ZIP 压缩的资源。
- CSV 文件可以存在数据库服务器上，通过 file:///URL 来访问。LOAD CSV 也支持通过 HTTPS、HTTP 和 FTP 来访问 CSV 文件。
- LOAD CSV 支持 HTTP 重定向，但基于安全考虑，重定向时不能改变协议类型，比如从 HTTPS 重定向到 HTTP。

文件 URL 的配置项

- dbms.security.allow_csv_import_from_file_urls[9]

这个选项决定 Cypher 在使用 LOAD CSV 时是否支持使用 fille:/// URL 来加载数据。该 URL 唯一标识了数据库服务器文件系统上的文件。dbms.security.allow_csv_import_from_file_urls=false 将完全禁止 LOAD CSV 访问文件系统。

- dbms.directories.import[10]

设置 LOAD CSV 中 file:/// URL 中的根路径。这必须设置为数据库服务器上的文件系统的单个目录，它让所有的请求从 file:/// URL 加载时都使用根路径的相对路径（类似 UNIX 下的 chroot 操作）。默认值是 import，这是基于安全考虑阻止数据库访问标准的 import 之外的目录下的文件。将 dbms.directories.import 设置为空可以消除这个安全隐患，允许访问系统上的任何文件，但是不推荐这么做。

文件 URLs 将相对于 dbms.directories.import 来解析。例如，一个典型的 URL 类似 file:///myfile.csv 或者 file:///myproject/myfile.csv。

- 如果 dbms.directories.import 设置的是默认值 import，那么在 LOAD CSV 语句将分别从<NEO4J_HOME>/import/myfile.csv 和<NEO4J_HOME>/import/myproject/myfile.csv 中读取数据。
- 如果设置为/data/csv，上面的 LOAD CSV 中的 URL 将分别从/data/csv/myfile.csv 和 /data/csv/myproject/myfile.csv 中读取数据。

详情参见下面小节的例子。

3.3.6.1　CSV 文件格式

使用 LOAD CSV 导入的 CSV 文件必须满足如下要求：

- 字符编码为 UTF-8。

---

[9] http://neo4j.com/docs/operationsmanual/3.1/reference/#config_dbms.security.allow_csv_import_from_file_urls
[10] http://neo4j.com/docs/operations-manual/3.1/reference/#config_dbms.directories.import

- 行结束符与操作系统关联，如 unix 上为\n，windows 上为\r\n。
- 默认的字段终止符为。
- 字段终止符可以使用 LOAD CSV 中的 FIELDTERMINATOR 选项来修改。
- CSV 文件允许引号字符串，但读取数据的时候引号字符会被丢弃。
- 字符串的引号字符为双引号"。
- 转义字符为\。

3.3.6.2 从 CSV 文件导入数据

从 CSV 文件导入数据到 Neo4j，可以用 LOAD CSV 把数据加载到查询语句中。然后使用正常的 Cypher 更新语句将数据写入到数据库中。

artists.csv

"1","ABBA","1992"

"2","Roxette","1986"

"3","Europe","1979"

"4","The Cardigans","1992"

查询

```
LOAD CSV FROM ' http://neo4j.com/docs/developer-manual/3.1/csv/artists.csv'
AS line
CREATE (:Artist { name: line[1], year: toInt(line[2])})
```

CSV 文件中的每一行都创建一个标签为 Artist 的节点。CSV 文件中的另外两列分别设置为节点的属性。

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 4
Properties set: 8
Labels added: 4
```

3.3.6.3 导入包含文件头的 CSV 文件

当导入的 CSV 文件包含文件头时，可以把每一行看作一个 map，而不是字符串数组。

artists.csv

"Id","Name","Year"

"1","ABBA","1992"

"2","Roxette","1986"

"3","Europe","1979"

"4","The Cardigans","1992"

查询

```
LOAD CSV WITH HEADERS FROM 'http://neo4j.com/docs/developer-
manual/3.1/csv/artists-with-headers.csv' AS line
CREATE (:Artist { name: line.Name, year: toInt(line.Year)})
```

这时，文件的开始行包含列的名称。指定 WITH HEADERS 后，可以通过对应的列名来访问指定的字段。

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 4
Properties set: 8
Labels added: 4
```

3.3.6.4　导入自定义分隔符的 CSV 文件

CSV 文件的分隔符有时候不是逗号，而是其他分隔符。这时，可以使用 FIELDTERMINATOR 来指定。

```
artists-fieldterminator.csv
"1";"ABBA";"1992"
"2";"Roxette";"1986"
"3";"Europe";"1979"
"4";"The Cardigans";"1992"
```

查询

```
LOAD CSV FROM 'http://neo4j.com/docs/developer-manual/3.1/csv/artists-
fieldterminator.csv' AS line FIELDTERMINATOR ';'
CREATE (:Artist { name: line[1], year: toInt(line[2])})
```

本例中字段之间以分号分隔，因此，在 LOAD CSV 中使用了 FIELDTERMINATOR 自定义分隔符。

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 4
Properties set: 8
Labels added: 4
```

3.3.6.5　导入巨量数据

如果导入的 CSV 文件包含百万数量级的行，可以使用 USING PERIODIC COMMIT 来指导 Neo4j 每导入一定数量行之后就提交（Commit）一次。这样可避免在事务过程中耗费大量的内存。默认情况下，每 1000 行会提交一次。更多详情，请参见“3.3.6.6 设置提交频率”小节。

查询

```
USING PERIODIC COMMIT
LOAD CSV FROM 'http://neo4j.com/docs/developer-manual/3.1/csv/artists.csv' AS
line
CREATE (:Artist { name: line[1], year: toInt(line[2])})
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 4
Properties set: 8
Labels added: 4
```

3.3.6.6 设置提交频率

可以设置提交的频率，如本例中设置为 500 行。

查询

```
USING PERIODIC COMMIT 500
LOAD CSV FROM 'http://neo4j.com/docs/developer-manual/3.1/csv/artists.csv' AS
line
CREATE (:Artist { name: line[1], year: toInt(line[2])})
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 4
Properties set: 8
Labels added: 4
```

3.3.6.7 导入包含转义字符的数据

本例中同时包含了引用字符和转义字符。

artists-with-escaped-char.csv

"1","The ""Symbol""","1992"

查询

```
LOAD CSV FROM 'http://neo4j.com/docs/developer-manual/3.1/csv/artists-with-
escaped-char.csv' AS line
CREATE (a:Artist { name: line[1], year: toInt(line[2])})
RETURN a.name AS name, a.year AS year, length(a.name) AS length
```



这里的字符串用双引号括起来，同时，关注本例中字符串的长度。

结果

```
+----------------+------+--------+
| name           | year | length |
+----------------+------+--------+
| "The "Symbol"" | 1992 | 12     |
+----------------+------+--------+
1 row
Nodes created: 1
Properties set: 2
Labels added: 1
```

3.3.7　CREATE

CREATE 语句用于创建图元素：节点和关系。

3.3.7.1　创建节点

1. 创建单个节点

通过如下语句创建单个节点：

查询

```
CREATE (n)
```

除了影响到的节点数之外，这个查询什么也不返回。

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 1
```

2. 创建多个节点

创建多个节点，中间以逗号分隔。

查询

```
CREATE (n), (m)
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 2
```

3. 创建带有标签的节点

创建带有标签的节点采用如下格式：

查询

```
CREATE (n:Person)
```

这个查询什么也不返回。

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 1
Labels added: 1
```

4. 创建带有多个标签的节点

如下语句在创建节点的时候，为其添加了两个标签。

查询

```
CREATE (n:Person:Swedish)
```

这个查询什么也没返回。

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 1
Labels added: 2
```

5. 创建同时带有标签和属性的节点

当创建一个带有标签的节点时，同时也可以添加属性。

查询

```
CREATE (n:Person { name: 'Andres', title: 'Developer' })
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 1
Properties set: 2
Labels added: 1
```

6. 返回创建的节点

查询

```
CREATE (a { name: 'Andres' })
RETURN a
```

返回了新创建的节点。

结果

```
+-----------------------+
| a                     |
```

```
+------------------------+
| Node[0]{name:"Andres"} |
+------------------------+
1 row
Nodes created: 1
Properties set: 1
```

3.3.7.2　创建关系

1. 创建两个节点之间的关系

要创建两个节点之间的关系，先需要找到这两个节点。一旦节点加载后，就可以简单地创建两者之间的关系。

查询

```
MATCH (a:Person),(b:Person)
WHERE a.name = 'Node A' AND b.name = 'Node B'
CREATE (a)-[r:RELTYPE]->(b)
RETURN r
```

查询返回了创建的关系。

结果

```
+---------------+
| r             |
+---------------+
| :RELTYPE[0]{} |
+---------------+
1 row
Relationships created: 1
```

2. 创建关系并设置属性

给关系设置属性类似于创建节点时设置节点属性一样。注意：设置的属性值可以是任意表达式。

查询

```
MATCH (a:Person),(b:Person)
WHERE a.name = 'Node A' AND b.name = 'Node B'
CREATE (a)-[r:RELTYPE { name: a.name + '<->' + b.name }]->(b)
RETURN r
```

结果

```
+-------------------------------------+
| r                                   |
+-------------------------------------+
| :RELTYPE[0]{name:"Node A<->Node B"} |
+-------------------------------------+
1 row
Relationships created: 1
Properties set: 1
```

3.3.7.3 创建一个完整路径

当使用 CREATE 和模式时，模式中所有还不存在的部分都会被创建。

查询

```
CREATE p =(andres { name:'Andres' })-[:WORKS_AT]->(neo)<-[:WORKS_AT]-(michael
{ name: 'Michael' })
RETURN p
```

这个查询创建了三个节点和两个关系，然后将它赋值给一个路径（Path）变量并返回它。

结果

```
+--------------------------------------------------------------------------------+
| p                                                                              |
+--------------------------------------------------------------------------------+
|[Node[0]{name:"Andres"},:WORKS_AT[0]{},Node[1]{},:WORKS_AT[1]{},Node[2]{name: |
| "Michael"}]                                                                    |
+--------------------------------------------------------------------------------+
1 row
Nodes created: 3
Relationships created: 2
Properties set: 2
```

3.3.7.4 CREATE 中使用参数

可以使用 map 来创建图的实体。map 中的所有键值对都会被设置到创建的关系或节点上。在下面的例子中还给节点添加了一个 Person 标签。

参数

```
{
"props" : {
"name" : "Andres",
"position" : "Developer"
}
}
```

查询

```
CREATE (n:Person $props)
RETURN n
```

结果

```
+--------------------------------------------+
| n                                          |
+--------------------------------------------+
| Node[0]{name:"Andres",position:"Developer"} |
+--------------------------------------------+
1 row
Nodes created: 1
Properties set: 2
Labels added: 1
```

用属性参数创建多个节点

通过使用一个 Cypher 的 map 数组，它将为每个 map 创建一个节点。

参数

```
{
"props" : [ {
"name" : "Andres",
"position" : "Developer"
}, {
"name" : "Michael",
"position" : "Developer"
} ]
}
```

查询

```
UNWIND $props AS map
CREATE (n)
SET n = map
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 2
Properties set: 4
```

3.3.8 MERGE

MERGE 可以确保图数据库中存在某个特定的模式（Pattern）。如果该模式不存在，那就创建它。

3.3.8.1 介绍

MERGE 或者匹配已存在的节点并绑定到它，或者创建新的然后绑定到它。它有点像 MATCH 和 CREATE 的组合。另外，可以让你指定让某个数据存在，不管它是匹配到还是创建它。例如，可以指定图中必须包含一个特定 name 的 user 节点。如果不存在正确 name 的 user 节点，那么就会创建一个。

当在整个模式上使用 MERGE 时，要么是整个模式匹配到，要么是整个模式被创建。MERGE 不能部分地应用于模式，如果希望部分匹配，可以将模式拆分为多个 MERGE 语句。

MERGE 图例如图 3-11 所示。

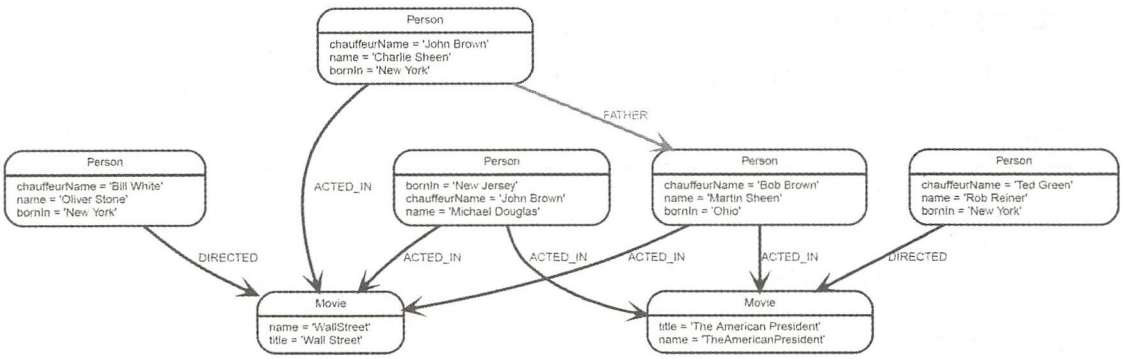


图 3-11　MERGE 图例

3.3.8.2　MERGE 节点

1. 合并带标签的节点

下面的例子合并给定标签的节点。

查询

```
MERGE (robert:Critic)
RETURN robert, labels(robert)
```

因为没有包含 Critic 标签的节点，一个新节点在数据库中创建了。

结果

```
+----------+----------------+
| robert   | labels(robert) |
+----------+----------------+
| Node[7]{} | ["Critic"]     |
+----------+----------------+
1 row
Nodes created: 1
Labels added: 1
```

2. 合并带多个属性的单个节点

合并有多个属性但并不是所有属性都匹配到已存在节点的单个节点。

查询

```
MERGE (charlie { name: 'Charlie Sheen', age: 10 })
RETURN charlie
```

名为'Charlie Sheen'的新节点创建了，因为不是所有属性都在已存在的'Charlie Sheen'节点上匹配到了。

结果

```
+--------------------------------------+
| charlie                              |
```

```
+-------------------------------------+
| Node[7]{name:"Charlie Sheen",age:10} |
+-------------------------------------+
1 row
Nodes created: 1
Properties set: 2
```

3. 合并同时指定标签和属性的节点

合并单个节点，要求它的标签和属性都能匹配到已存在的节点。

查询

```
MERGE (michael:Person { name: 'Michael Douglas' })
RETURN michael.name, michael.bornIn
```

'Michael Douglas'节点被匹配到，同时返回它的 name 和 bornIn 属性。

结果

```
+-------------------+----------------+
| michael.name      | michael.bornIn |
+-------------------+----------------+
| "Michael Douglas" | "New Jersey"   |
+-------------------+----------------+
1 row
```

4. 合并属性来自已存在节点的单个节点

当每个绑定节点的属性‘p’来自一个节点集时，如果‘p’存在重复，创建的时候只会创建一次。

查询

```
MATCH (person:Person)
MERGE (city:City { name: person.bornIn })
RETURN person.name, person.bornIn, city
```

本例中创建了三个 City 节点，它们的 name 属性分别为"New York"、"Ohio"和 New Jersey。注意：尽管 MATCH 匹配的结果总有三个节点的 bornIn 属性值都为"New York"，但只创建了一个"New York"节点。因为第一次匹配的时候，"New York"未匹配到，因此创建一个。然而，新创建的"New York"被第二个和第三个匹配到了。

结果

```
+-------------------+----------------+---------------------------+
| person.name       | person.bornIn  | city                      |
+-------------------+----------------+---------------------------+
| "Rob Reiner"      |   "New York"   | Node[7]{name:"New York"}  |
| "Oliver Stone"    |   "New York"   | Node[7]{name:"New York"}  |
| "Charlie Sheen"   |   "New York"   | Node[7]{name:"New York"}  |
| "Michael Douglas" |   "New Jersey" | Node[8]{name:"New Jersey"} |
| "Martin Sheen"    |   "Ohio"       | Node[9]{name:"Ohio"}      |
+-------------------+----------------+---------------------------+
5 rows
Nodes created: 3
```

```
Properties set: 3
Labels added: 3
```

3.3.8.3 MERGE 在 CREATE 和 MATCH 中的使用

1. MERGE 与 CREATE 搭配

检查节点是否存在，如果不存在则创建它并设置属性。

查询

```
MERGE (keanu:Person { name: 'Keanu Reeves' })
ON CREATE SET keanu.created = timestamp()
RETURN keanu.name, keanu.created
```

本查询创建了 keanu 节点，并设置 created 属性为创建的时间戳。

结果

```
+----------------+---------------+
| keanu.name     | keanu.created |
+----------------+---------------+
| "Keanu Reeves" | 1486061063894 |
+----------------+---------------+
1 row
Nodes created: 1
Properties set: 2
Labels added: 1
```

2. MERGE 与 MATCH 搭配

匹配节点，并在找到的节点上设置属性。

查询

```
MERGE (person:Person)
ON MATCH SET person.found = TRUE RETURN person.name, person.found
```

本查询找到所有的 Person 节点，并设置 found 属性为 true，然后返回它们。

结果

```
+-------------------+--------------+
| person.name       | person.found |
+-------------------+--------------+
| "Rob Reiner"      | true         |
| "Oliver Stone"    | true         |
| "Charlie Sheen"   | true         |
| "Michael Douglas" | true         |
| "Martin Sheen"    | true         |
+-------------------+--------------+
5 rows
Properties set: 5
```

3. MERGE 与 CREATE 和 MATCH 同时使用

检查节点是否存在，如果不存在，则创建它并设置属性。

查询

```
MERGE (keanu:Person { name: 'Keanu Reeves' })
ON CREATE SET keanu.created = timestamp()
ON MATCH SET keanu.lastSeen = timestamp()
RETURN keanu.name, keanu.created, keanu.lastSeen
```

本查询创建'keanu'节点并设置 created 属性值为创建的时间戳。如果'keanu'已经存在，将为它设置一个新属性'lastSeen'。也就是说，当'keanu'不存在时，创建后的'keanu'节点将没有'lastSeen'属性。

结果

```
+----------------+---------------+----------------+
| keanu.name     | keanu.created | keanu.lastSeen |
+----------------+---------------+----------------+
| "Keanu Reeves" | 1486061067497 | <null>         |
+----------------+---------------+----------------+
1 row
Nodes created: 1
Properties set: 2
Labels added: 1
```

4. 利用 MERGE 和 MATCH 设置多属性

如果需要设置多个属性，将它们简单地以逗号分开即可。

查询

```
MERGE (person:Person)
ON MATCH SET person.found = TRUE , person.lastAccessed = timestamp()
RETURN person.name, person.found, person.lastAccessed
```

结果

```
+------------------+--------------+---------------------+
| person.name      | person.found | person.lastAccessed |
+------------------+--------------+---------------------+
| "Rob Reiner"     | true         | 1486061066390       |
| "Oliver Stone"   | true         | 1486061066390       |
| "Charlie Sheen"  | true         | 1486061066390       |
| "Michael Douglas"| true         | 1486061066390       |
| "Martin Sheen"   | true         | 1486061066390       |
+------------------+--------------+---------------------+
5 rows
Properties set: 10
```

3.3.8.4　MERGE 关系

MERGE 可用于匹配或者创建关系。

查询

```
MATCH (charlie:Person { name: 'Charlie Sheen' }),(wallStreet:Movie { title:
'Wall Street' })
MERGE (charlie)-[r:ACTED_IN]->(wallStreet)
```

```
RETURN charlie.name, type(r), wallStreet.title
```

因为"Charlie Sheen"已经在"Wall Street"参演了，所以找到已存的关系并返回。注意：使用 MERGE 去匹配或者创建关系时，必须至少指定一个绑定的节点。

结果

```
+-----------------+------------+------------------+
| charlie.name    | type(r)    | wallStreet.title |
+-----------------+------------+------------------+
| "Charlie Sheen" | "ACTED_IN" | "Wall Street"    |
+-----------------+------------+------------------+
1 row
```

1. 合并多个关系

当 MERGE 应用于整个模式时，要么全部匹配上，要么全部新创建。

查询

```
MATCH (oliver:Person { name: 'Oliver Stone' }),(reiner:Person { name: 'Rob
Reiner' })
MERGE (oliver)-[:DIRECTED]->(movie:Movie)<-[:ACTED_IN]-(reiner)
RETURN movie
```

在本例中，'Oliver Stone'和'Rob Reiner'未一起工作过。当试图在他们之间合并一个电影连接时，Neo4j 不会使用任何已存在的电影，而是创建一个新的'movie'节点。

结果

```
+-----------+
| movie     |
+-----------+
| Node[7]{} |
+-----------+
1 row
Nodes created: 1
Relationships created: 2
Labels added: 1
```

2. 合并无方向关系

MERGE 也可以用于合并无方向的关系。当需要创建一个关系的时候，它将选择一个任意的方向。

查询

```
MATCH (charlie:Person { name: 'Charlie Sheen' }),(oliver:Person { name:
'Oliver Stone' })
MERGE (charlie)-[r:KNOWS]-(oliver)
RETURN r
```

因为'Charlie Sheen'和'Oliver Stone'相互不认识，所以 MERGE 查询将在他们之间创建一个 KNOWS 关系。创建的关系的方向是任意的。

结果

```
+-------------+
| r           |
+-------------+
| :KNOWS[8]{} |
+-------------+
1 row
Relationships created: 1
```

3. 合并已存在两节点之间的关系

MERGE 可用于连接前面的 MATCH 和 MERGE 语句，在两个绑定的节点 m 和 n 上创建一个关系。m 节点是 MATCH 语句返回的，而 n 节点是前面的 MERGE 语句创建或者匹配到的。

查询

```
MATCH (person:Person)
MERGE (city:City { name: person.bornIn })
MERGE (person)-[r:BORN_IN]->(city)
RETURN person.name, person.bornIn, city
```

这个例子来自“3.3.8 MERGE”小节。第二个 MERGE 在每个人和他的 bornIn 属性对应的城市之间创建了一个 BORN_IN 关系。"Charlie Sheen"、"Rob Reiner"和"Oliver Stone"与同一个城市节点（"New York"）都有一个 BORIN_IN 关系。

结果

```
+------------------+--------------+---------------------------+
| person.name      | person.bornIn | city                      |
+------------------+--------------+---------------------------+
| "Rob Reiner"     | "New York"   | Node[7]{name:"New York"}  |
| "Oliver Stone"   | "New York"   | Node[7]{name:"New York"}  |
| "Charlie Sheen"  | "New York"   | Node[7]{name:"New York"}  |
| "Michael Douglas" | "New Jersey" | Node[8]{name:"New Jersey"} |
| "Martin Sheen"   | "Ohio"       | Node[9]{name:"Ohio"}      |
+------------------+--------------+---------------------------+
5 rows
Nodes created: 3
Relationships created: 5
Properties set: 3
Labels added: 3
```

4. 合并一个已存在节点和一个合并的节点之间的关系

MERGE 能够同时创建一个新节点 n 和一个已存在节点 m 与 n 之间的关系。

查询

```
MATCH (person:Person)
MERGE (person)-[r:HAS_CHAUFFEUR]->(chauffeur:Chauffeur { name:
person.chauffeurName })
RETURN person.name, person.chauffeurName, chauffeur
```

在本例中，MERGE 未匹配到，这里没有标签为 Chauffeur 的节点和 HAS_CHAUFFUR 关系。MERGE 创建了 5 个带有 Chauffeur 标签的节点，每个节点的包含一个 name 属性，属

性的值来自每个匹配到的 Person 节点的 chauffeurName 属性的值。MERGE 同时还在每个 Person 节点与新创建的 Chauffeur 节点之间创建了一个 HAS_CHAUFFEUR 关系。

结果

```
+------------------+---------------------+----------------------------+
| person.name      | person.chauffeurName | chauffeur                  |
+------------------+---------------------+----------------------------+
| "Rob Reiner"     | "Ted Green"          | Node[7]{name:"Ted Green"}  |
| "Oliver Stone"   | "Bill White"         | Node[8]{name:"Bill White"} |
| "Charlie Sheen"  | "John Brown"         | Node[9]{name:"John Brown"} |
| "Michael Douglas" | "John Brown"        | Node[10]{name:"John Brown"} |
| "Martin Sheen"   | "Bob Brown"          | Node[11]{name:"Bob Brown"} |
+------------------+---------------------+----------------------------+
5 rows
Nodes created: 5
Relationships created: 5
Properties set: 5
Labels added: 5
```

3.3.8.5 用 MERGE 的唯一性约束

当使用的模式涉及唯一性约束时，Cypher 可以通过 MERGE 来防止获取相冲突的结果。在这种情况下，至多有一个节点匹配该模式。例如，给定两个唯一性约束:Person(id)和:Person(ssn)，如果存在两个不同的节点分别 id 为 12 和 ssn 为 437 或者只有一个节点有其中一个属性，那么这样的查询 MERGE (n:Person {id: 12, ssn: 437})将失败。

下面的例子分别在 Person 的 name 和 role 属性上创建一个唯一性约束。

```
CREATE CONSTRAINT ON (n:Person) ASSERT n.name IS UNIQUE;
CREATE CONSTRAINT ON (n:Person) ASSERT n.role IS UNIQUE;
```

如果节点未找到，使用唯一性约束创建该节点。

查询

```
ERGE (laurence:Person { name: 'Laurence Fishburne' })
RETURN laurence.name
```

本查询创建了'laurence'节点。如果'laurence'已经存在，MERGE 仅仅匹配已经存在的节点。

结果

```
+----------------------+
| laurence.name        |
+----------------------+
| "Laurence Fishburne" |
+----------------------+
1 row
Nodes created: 1
Properties set: 1
Labels added: 1
```

使用唯一性约束匹配已存在的节点。

查询

```
MERGE (oliver:Person { name: 'Oliver Stone' })
RETURN oliver.name, oliver.bornIn
```

'oliver'节点已经存在了，因此 MERGE 只是匹配它。

结果

```
+----------------+---------------+
| oliver.name    | oliver.bornIn |
+----------------+---------------+
| "Oliver Stone" | "New York"    |
+----------------+---------------+
1 row
```

1. 唯一性约束与部分匹配

当只有部分匹配时，使用唯一性约束合并将失败。

查询

```
MERGE (michael:Person { name: 'Michael Douglas', role: 'Gordon Gekko' })
RETURN michael
```

这里有一个唯一匹配到的 name 为'Michael Douglas'的节点，但没有具有唯一的 role 属性为'Gordon Gekko'的节点，因此 MERGE 匹配失败。

错误消息

```
Merge did not find a matching node michael and can not create a new node due to conflicts with existing unique nodes
```

2. 唯一性约束与匹配冲突

当有匹配的冲突结果时，使用 MERGE 唯一性约束将失败。

查询

```
MERGE (oliver:Person { name: 'Oliver Stone', role: 'Gordon Gekko' })
RETURN oliver
```

错误消息

```
Merge did not find a matching node oliver and can not create a new node due to conflicts with existing unique nodes
```

3.3.8.6　使用 map 参数

MERGE 不支持像 CREATE 节点时那样使用 map 参数。要在 MERGE 中使用 map 参数，需要显式地使用希望用的属性。如下例所示。

参数

```
{
"param" : {
```

```
"name" : "Keanu Reeves",
"role" : "Neo"
}
}
```

查询

```
MERGE (person:Person { name: $param.name, role: $param.role })
RETURN person.name, person.role
```

结果

```
+----------------+-------------+
| person.name    | person.role |
+----------------+-------------+
| "Keanu Reeves" | "Neo"       |
+----------------+-------------+
1 row
Nodes created: 1
Properties set: 2
Labels added: 1
```

3.3.9 SET

SET 语句用于更新节点的标签以及节点和关系的属性。

SET 可以使用 map 中的参数来设置属性。



设置节点的标签是幂等性操作，即如果试图设置一个已经存在的标签到节点上，什么也不会发生。查询统计会自己判断是否需要处理。

SET 图例如图 3-12 所示。

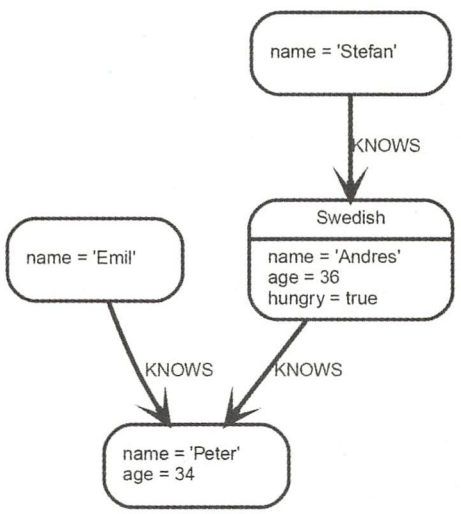


图 3-12　SET 图例

3.3.9.1　设置属性

可以使用 SET 设置节点或者关系的属性。

查询

```
MATCH (n { name: 'Andres' })
SET n.surname = 'Taylor'
RETURN n
```

返回最新修改过的节点。

结果

```
+-------------------------------------------------------------+
| n                                                           |
+-------------------------------------------------------------+
| Node[0]{surname:"Taylor",name:"Andres",age:36,hungry:true} |
+-------------------------------------------------------------+
1 row
Properties set: 1
```

3.3.9.2　删除属性

通常使用 REMOVE 来删除一个属性，但有时候也可以随手用 SET。将属性设置为 null 将删除该属性。如下所示：

查询

```
MATCH (n { name: 'Andres' })
SET n.name = NULL RETURN n
```

设置 name 属性为空之后，name 属性被删除了。

结果

```
+-----------------------------+
| n                           |
+-----------------------------+
| Node[0]{hungry:true,age:36} |
+-----------------------------+
1 row
Properties set: 1
```

3.3.9.3　在节点和关系间复制属性

可以使用 SET 复制一个图元素的所有属性到另外一个图元素。记住，这样做会删除目标元素的所有其他属性。

查询

```
MATCH (at { name: 'Andres' }),(pn { name: 'Peter' })
SET at = pn
RETURN at, pn
```

“Andres”节点的所有属性被“Peter”节点的属性替换了。

结果

```
+-----------------------------+-----------------------------+
| at                          | pn                          |
+-----------------------------+-----------------------------+
| Node[0]{name:"Peter",age:34} | Node[2]{name:"Peter",age:34} |
+-----------------------------+-----------------------------+
1 row
Properties set: 3
```

3.3.9.4 从 map 中添加属性

当用 map 来设置属性时，可以使用+=形式的 SET，用于只添加属性，而不删除图元素中已存在的属性。

查询

```
MATCH (peter { name: 'Peter' })
SET peter += { hungry: TRUE , position: 'Entrepreneur' }
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Properties set: 2
```

3.3.9.5 使用参数设置属性

可以使用参数来给属性赋值。

参数

```
{
"surname" : "Taylor"
}
```

查询

```
MATCH (n { name: 'Andres' })
SET n.surname = $surname
RETURN n
```

“Andres”节点新增一个 surname 属性。

结果

```
+-------------------------------------------------------------+
| n                                                           |
+-------------------------------------------------------------+
| Node[0]{surname:"Taylor",name:"Andres",age:36,hungry:true} |
+-------------------------------------------------------------+
1 row
Properties set: 1
```

3.3.9.6　使用一个参数设置所有属性

使用参数提供的属性集合来设置节点的属性时，该节点已存在的所有属性将被替换掉。

参数

```
{
"props" : {
"name" : "Andres",
"position" : "Developer"
}
}
```

查询

```
MATCH (n { name: 'Andres' })
SET n = $props
RETURN n
```

“Andres”节点的所有属性都被替换为 props 参数中的属性了。

结果

```
+--------------------------------------------+
| n                                          |
+--------------------------------------------+
| Node[0]{name:"Andres",position:"Developer"} |
+--------------------------------------------+
1 row
Properties set: 4
```

3.3.9.7　使用一个 SET 语句设置多个属性

如果想一次设置多个属性，简单地用逗号分开即可。

查询

```
MATCH (n { name: 'Andres' })
SET n.position = 'Developer', n.surname = 'Taylor'
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Properties set: 2
```

3.3.9.8　设置节点的标签

用 SET 可给节点设置标签。

查询

```
MATCH (n { name: 'Stefan' })
SET n :German
RETURN n
```

查询返回新增了标签的节点。

结果

```
+------------------------+
| n                      |
+------------------------+
| Node[3]{name:"Stefan"} |
+------------------------+
1 row
Labels added: 1
```

3.3.9.9 给一个节点设置多个标签

使用 SET 给一个节点设置多个标签时，不同的标签之间用冒号分隔。

查询

```
MATCH (n { name: 'Emil' })
SET n :Swedish:Bossman
RETURN n
```

查询返回新增多个标签的节点。

结果

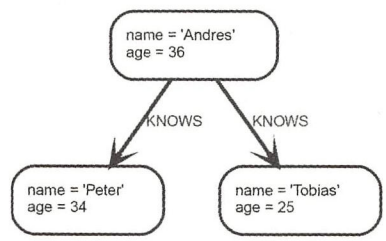
```
+----------------------+
| n                    |
+----------------------+
| Node[1]{name:"Emil"} |
+----------------------+
1 row
Labels added: 2
```

3.3.10 DELETE

DELETE 语句用于删除图元素（节点、关系或路径）。

删除属性和标签，参见“3.3.11 REMOVE”小节。记住，不能只删除节点，而不删除与之相连的关系。要么显式地删除对应的关系，要么使用 DETACH DELETE。

DELETE 图例如图 3-13 所示。



3-13 DELETE 图例

3.3.10.1　删除单个节点

使用 DELETE 删除一个节点。

查询

```
MATCH (n:Useless)
DELETE n
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes deleted: 1
```

3.3.10.2　删除所有节点和关系

这个查询适用于删除少量数据，但不适用于删除巨量数据。

查询

```
MATCH (n)
DETACH DELETE n
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes deleted: 3
Relationships deleted: 2
```

3.3.10.3　删除一个节点及其所有的关系

当需要删除一个节点及与该节点的所有关系时，可用 DETACH DELETE。

查询

```
MATCH (n { name: 'Andres' })
DETACH DELETE n
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes deleted: 1
Relationships deleted: 2
```

3.3.11　REMOVE

REMOVE 语句用于删除图元素的属性和标签。对于删除节点和关系，参见“3.3.10 DELETE”小节。



删除节点的标签是幂等性操作。如果删除一个节点的不存在的标签，什么也不会发生。查询统计会自己判断是否需要处理。

REMOVE 图例如图 3-14 所示。

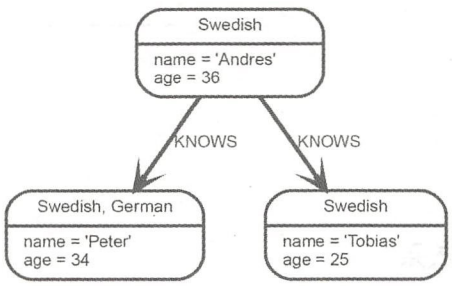


图 3-14　REMOVE 图例

3.3.11.1　删除一个属性

Neo4j 不允许属性存储空值 null。如果属性的值不存在，那么节点或者关系中的属性将被删除。这也可以通过 REMOVE 来删除。

查询

```
MATCH (andres { name: 'Andres' })
REMOVE andres.age
RETURN andres
```

返回的节点没有 age 属性了。

结果

```
+------------------------+
| andres                 |
+------------------------+
| Node[0]{name:"Andres"} |
+------------------------+
1 row
Properties set: 1
```

3.3.11.2　删除节点的一个标签

可用 REMOVE 删除一个标签。

查询

```
MATCH (n { name: 'Peter' })
REMOVE n:German
RETURN n
```

结果

```
+------------------------------+
| n                            |
+------------------------------+
| Node[2]{name:"Peter",age:34} |
+------------------------------+
1 row
Labels removed: 1
```

3.3.11.3　删除节点的多个标签

可以使用 REMOVE 删除多个标签。

查询

```
MATCH (n { name: 'Peter' })
REMOVE n:German:Swedish
RETURN n
```

结果

```
+------------------------------+
| n                            |
+------------------------------+
| Node[2]{name:"Peter",age:34} |
+------------------------------+
1 row
Labels removed: 2
```

3.3.12　FOREACH

FOREACH 语句用于更新列表中的数据，或者来自路径的组件，或者来自聚合的结果。

列表（Lists）和路径（Paths）是 Cypher 中的关键概念，可以使用 FOREACH 来更新其中的数据。它可以在路径或者聚合的列表的每个元素上执行更新命令。FOREACH 括号中的变量是与外部分开的，这意味着 FOREACH 中创建的变量不能用于该语句之外。

在 FOREACH 括号内，可以执行任何的更新命令，包括 CREATE、CREATE UNIQUE、DELETE 和 FOREACH。如果希望对列表中的每个元素执行额外的 MATCH 命令，使用 UNWIND 命令更合适。FOREACH 图例如图 3-15 所示。

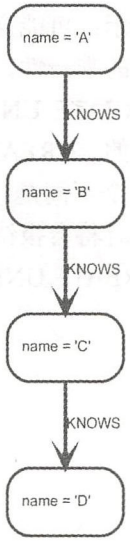


图 3-15　FOREACH 图例

3.3.12.1　标记路径上的所有节点

这个查询将设置路径上所有节点的 marked 属性为 true 值。

查询

```
MATCH p =(begin)-[*]->(END )
WHERE begin.name = 'A' AND END .name = 'D'
FOREACH (n IN nodes(p)| SET n.marked = TRUE )
```

本查询无返回，但设置了 4 个属性。

结果

```
+-------------------+
| No data returned. |
+-------------------+
Properties set: 4
```

3.3.12.2 从列表中创建朋友

下面的查询将列表中的人全部加为'A'的朋友。

```
MATCH (a:Person {name:"A"})
FOREACH (name IN ["Mike", "Carl", "Bruce"] |
CREATE (a)-[:FRIEND]->(:Person {name: name}))
```

本查询无返回，但创建了三个节点，每个节点添加了标签 Person 和设置了 name 属性，并将这三个节点与“A”之间建立 FRIEND 关系。

3.3.13 CREATE UNIQUE

CREATE UNIQUE 语句相当于 MATCH 和 CREATE 的混合体——尽可能地匹配，然后创建未匹配到的。

3.3.13.1 简介

提示：可能会想到用 MERGE 来代替 CREATE UNIQUE，然而 MERGE 并不能很强地保证关系的唯一性。

CREATE UNIQUE 介于 MATCH 和 CREATE 之间，其作用是匹配所能匹配得上的，创建不存在的。CREATE UNIQUE 尽可能地减少对图的改变，充分利用已有的图。与 MATCH 的另外一个不同是，CREATE UNIQUE 假设模式是唯一性的，如果有多个匹配的子图可以找到，此时将会报错。

CREATE UNIQUE 图例如图 3-16 所示。

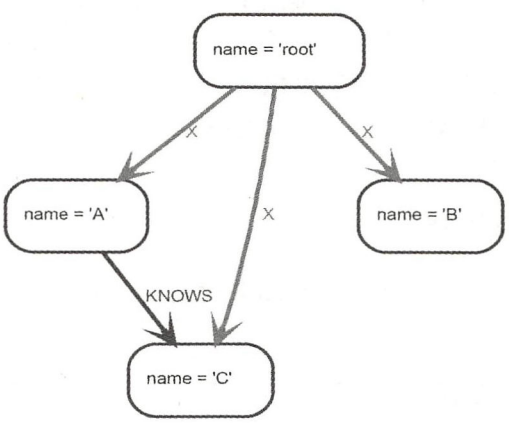


图 3-16 CREATE UNIQUE 图例

3.3.13.2 创建唯一节点

1. 创建未匹配到的节点

如果模式描述的节点未匹配到，则创建一个新节点。

查询

```
MATCH (root { name: 'root' })
CREATE UNIQUE (root)-[:LOVES]-(someone)
RETURN someone
```

root 节点没有任何 LOVES 关系。因此，创建了一个节点及其与 root 节点的 LOVES 关系。

结果

```
+-----------+
| someone   |
+-----------+
| Node[4]{} |
+-----------+
1 row
Nodes created: 1
Relationships created: 1
```

2. 用含值的模式创建节点

描述的模式可以在节点中包含值，语法如下：prop: <expression>。

查询

```
MATCH (root { name: 'root' })
CREATE UNIQUE (root)-[:X]-(leaf { name: 'D' })
RETURN leaf
```

没有与 root 节点相连的 name 为 D 的节点，所以创建一个新的节点来匹配该模式。

结果

```
+-------------------+
| leaf              |
+-------------------+
| Node[4]{name:"D"} |
+-------------------+
1 row
Nodes created: 1
Relationships created: 1
Properties set: 1
```

3. 创建未匹配到带标签的节点

如果描述的模式需要一个带标签的节点，而数据库中没有带给定标签的节点，Cypher 将创建一个新的节点。

查询

```
MATCH (a { name: 'A' })
CREATE UNIQUE (a)-[:KNOWS]-(c:blue)
```

```
RETURN c
```

与'A'节点相连的 KNOWS 关系有一个'C'节点，但'C'节点没有 blue 标签。所以创建了一个带有 blue 标签的节点和从'A'到它的 KNOWS 关系。

结果

```
+-----------+
| c         |
+-----------+
| Node[4]{} |
+-----------+
1 row
Nodes created: 1
Relationships created: 1
Labels added: 1
```

3.3.13.3 创建唯一关系

1. 创建未匹配到的关系

CREATE UNIQUE 用于描述应该被找到的或需要创建的模式。

查询

```
MATCH (lft { name: 'A' }),(rgt)
WHERE rgt.name IN ['B', 'C']
CREATE UNIQUE (lft)-[r:KNOWS]->(rgt)
RETURN r
```

匹配一个左节点和两个右节点之间的关系。其中一个关系已存在，因此能匹配到，然后创建了不存在的关系。

结果

```
+-------------+
| r           |
+-------------+
| :KNOWS[4]{} |
| :KNOWS[3]{} |
+-------------+
2 rows
Relationships created: 1
```

2. 用含值的模式创建关系

创建模式中含值的关系。

查询

```
MATCH (root { name: 'root' })
CREATE UNIQUE (root)-[r:X { since: 'forever' }]-()
RETURN r
```

本例中希望关系有一个值，因为没有这样的关系匹配到，因此创建了一个新节点和关系。注意，因为不关心创建的节点，所以没有对该节点命名。

结果

```
+------------------------+
| r                      |
+------------------------+
| :X[4]{since:"forever"} |
+------------------------+
1 row
Nodes created: 1
Relationships created: 1
Properties set: 1
```

3.3.13.4　描述复杂模式

就像 MATCH 和 CREATE 语句一样，CREATE UNIQUE 描述的模式也可以用逗号分隔。

查询

```
MATCH (root { name: 'root' })
CREATE UNIQUE (root)-[:FOO]->(x),(root)-[:BAR]->(x)
RETURN x
```

本例中的模式使用了两条用逗号分隔的路径。

结果

```
+-----------+
| x         |
+-----------+
| Node[4]{} |
+-----------+
1 row
Nodes created: 1
Relationships created: 2
```

3.3.14　RETURN

RETURN 语句定义了查询结果集中返回的内容。

在查询的 RETURN 部分定义了模式中感兴趣的部分。它可以是节点、关系或者是它们的属性。RETURN 图例如图 3-17 所示。



如果只需要属性值，要尽量避免返回整个节点或关系，这样有助于提高性能。

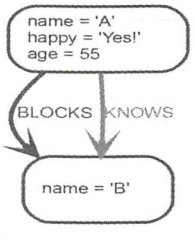


图 3-17　RETURN 图例

3.3.14.1 返回节点

返回匹配到的节点，如下所示。

查询

```
MATCH (n { name: 'B' })
RETURN n
```

本例中返回包含 name 属性值为'B'的节点。

结果

```
+-------------------+
| n                 |
+-------------------+
| Node[1]{name:"B"} |
+-------------------+
1 row
```

3.3.14.2 返回关系

返回匹配的关系，如下所示。

查询

```
MATCH (n { name: 'A' })-[r:KNOWS]->(c)
RETURN r
```

本例中返回了关系。

结果

```
+-------------+
| r           |
+-------------+
| :KNOWS[0]{} |
+-------------+
1 row
```

3.3.14.3 返回属性

返回属性可以用点来引用属性，如下所示。

查询

```
MATCH (n { name: 'A' })
RETURN n.name
```

本例中返回了 name 属性的值。

结果

```
+--------+
| n.name |
+--------+
| "A"    |
+--------+
1 row
```

3.3.14.4　返回所有元素

当希望返回查询中找到的所有节点、关系和路径时，可以使用星号*表示。如下所示。

查询

```
MATCH p =(a { name: 'A' })-[r]->(b)
RETURN *
```

本例中返回了两个节点、关系和路径。

结果

```
+-------------------+-------------+-----------------------------------+---------+
| a                 | b           | p                                 | r       |
+-------------------+-------------+-----------------------------------+---------+
| Node[0]{name:"A", | Node[1]     |[Node[0]{name:"A",                 |:BLOCKS[1]|
|  happy:"Yes!",    |  {name:"B"} | happy:"Yes!",age:55},             |  {}     |
|  age:55}          |             | :BLOCKS[1]{},Node[1]{name:"B"}]   |         |
+-------------------+-------------+-----------------------------------+---------+
| Node[0]{name:"A", | Node[1]     |[Node[0]{name:"A",| :KNOWS[0]{}    |         |
|  happy:"Yes!",    |  {name:"B"} | happy:"Yes!",age:55},             |         |
|  age:55}          |             | :KNOWS[0]{},Node[1]{name:"B"}]    |         |
+-------------------+-------------+-----------------------------------+---------+
2 rows
```

3.3.14.5　变量中的特殊字符

如果想使用空格等特殊字符，可以用反引号“`”将其括起来，如下所示。

查询

```
MATCH (`This isn't a common variable`)
WHERE `This isn't a common variable`.name = 'A'
RETURN `This isn't a common variable`.happy
```

返回 name 属性值为'A'的节点。

结果

```
+-------------------------------------+
| `This isn't a common variable`.happy |
+-------------------------------------+
| "Yes!"                              |
+-------------------------------------+
1 row
```

3.3.14.6　列别名

如果希望列名不同于表达式中使用的名字，可以使用 AS<new name>对其重命名。

查询

```
MATCH (a { name: 'A' })
RETURN a.age AS SomethingTotallyDifferent
```

返回节点的 age 属性，但列名由原有的 age 重命名为 SomethingTotallyDifferent 了。

结果

```
+---------------------------+
| SomethingTotallyDifferent |
+---------------------------+
| 55                        |
+---------------------------+
1 row
```

3.3.14.7 可选属性

如果某个属性可能存在，也可能不存在。这时，依然可以正常地去查询，对于不存在的属性，Cypher 返回 null。

查询

```
MATCH (n)
RETURN n.age
```

本例中有 age 属性的节点返回了具体的年龄，而没有这个属性的节点则返回 null。

结果

```
+--------+
| n.age  |
+--------+
| 55     |
| <null> |
+--------+
2 rows
```

3.3.14.8 其他表达式

任何表达式都可以作为返回项，如字面值、断言、属性、函数和任何其他表达式。

查询

```
MATCH (a { name: 'A' })
RETURN a.age > 30, "I'm a literal",(a)-->()
```

本例中返回了断言、字符串和带模式表达参数的函数调用。

结果

```
+------------+-----------------+-------------------------------------------+
| a.age > 30 | "I'm a literal" | (a)-->()                                  |
+------------+-----------------+-------------------------------------------+
| true       | "I'm a literal" | [[Node[0]{name:"A",happy:"Yes!",age:55},  |
|            |                 |   :BLOCKS[1]{},Node[1]{name:"B"}],        |
|            |                 |   [Node[0]{name:"A",happy:"Yes!",age:55}, |
|            |                 |   :KNOWS[0]{},Node[1]{name:"B"}]]         |
+------------+-----------------+-------------------------------------------+
1 row
```

3.3.14.9 唯一性结果

DISTINCT 用于仅仅获取结果集中所依赖列的唯一行。

查询

```
MATCH (a { name: 'A' })-->(b)
RETURN DISTINCT b
```

本例中只返回了一次 name 为 B 的节点。

结果

```
+--------------------+
| b                  |
+--------------------+
| Node[1]{name:"B"} |
+--------------------+
1 row
```

3.3.15　ORDER BY

ORDER BY 是紧跟 RETURN 或者 WITH 的子句，它指定了输出的结果应该如何排序。



不能对节点或关系进行排序，只能对它们的属性进行排序。ORDER BY 依赖值的比较来排序，具体可参见“3.2.5.8 值的排序与比较”小节。

在变量的范围方面，ORDER BY 遵循特定的规则，这取决于 RETURN 的投射或 WITH 语句是否聚合或者 DISTINCT。如果它是一个聚合或者 DISTINCT 投射，那么只有投射中的变量可用。如果投射不修改输出基数（聚合和 DISTINCT 做的），在投射之前可用的变量也可以用。当投射语句覆盖了已存在的变量时，只有新的变量可用。如图 3-18 所示为 ORDER BY 图例。

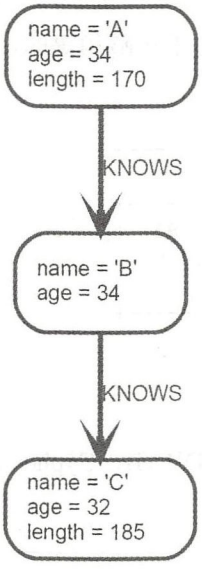


图 3-18　ORDER BY 图例

仅供非商业用途或交流学习使用

3.3.15.1 根据属性对节点进行排序

ORDER BY 用于对输出进行排序。

查询

```
MATCH (n)
RETURN n
ORDER BY n.name
```

返回了根据节点 name 属性进行排序的节点序列。

结果

```
+-----------------------------------+
| n                                 |
+-----------------------------------+
| Node[0]{name:"A",age:34,length:170} |
| Node[1]{name:"B",age:34}          |
| Node[2]{name:"C",age:32,length:185} |
+-----------------------------------+
3 rows
```

3.3.15.2 根据多个属性对节点进行排序

ORDER BY 语句中支持根据多个属性对节点进行排序。Cypher 将先根据第一个变量进行排序，对于相等的值，然后再检查 ORDER BY 中的下一个属性值，依次类推。

查询

```
MATCH (n)
RETURN n
ORDER BY n.age, n.name
```

本例中先根据他们的年龄排序，对于年龄相等的，再根据他们的名字来排序。

结果

```
+-----------------------------------+
| n                                 |
+-----------------------------------+
| Node[2]{name:"C",age:32,length:185} |
| Node[0]{name:"A",age:34,length:170} |
| Node[1]{name:"B",age:34}          |
+-----------------------------------+
3 rows
```

3.3.15.3 节点降序排列

在排序的变量后面添加 DESC[ENDING]，Cypher 将以逆序（即降序）对输出进行排序。

查询

```
MATCH (n)
RETURN n
ORDER BY n.name DESC
```


仅供非商业用途或交流学习使用

本例中根据节点的 name 降序排列这些节点。

结果

```
+-------------------------------------+
| n                                   |
+-------------------------------------+
| Node[2]{name:"C",age:32,length:185} |
| Node[1]{name:"B",age:34}            |
| Node[0]{name:"A",age:34,length:170} |
+-------------------------------------+
3 rows
```

3.3.15.4　空值的排序

当结果集中包含 null 值时，对于升序排列，null 总是在结果集的末尾。而对于降序排序，null 值总是排在最前面。

查询

```
MATCH (n)
RETURN n.length, n
ORDER BY n.length
```

本例中返回了以 length 属性排序的节点，没有 length 属性的节点排在最后。

结果

```
+----------+-------------------------------------+
| n.length | n                                   |
+----------+-------------------------------------+
| 170      | Node[0]{name:"A",age:34,length:170} |
| 185      | Node[2]{name:"C",age:32,length:185} |
| <null>   | Node[1]{name:"B",age:34}            |
+----------+-------------------------------------+
3 rows
```

3.3.16　LIMIT

LIMIT 限制输出的行数。LIMIT 可接受结果为正整数的任意表达式，但表达式不能引用节点或者关系。LIMIT 图例如图 3-19 所示。

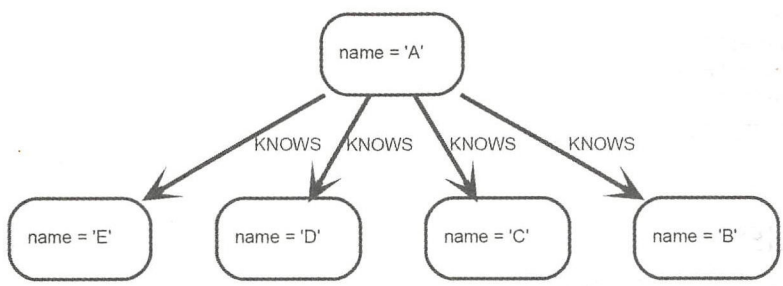


图 3-19　LIMIT 图例

仅供非商业用途或交流学习使用

3.3.16.1 返回开始部分

从最开始返回结果的一个子集，语法如下：

查询

```
MATCH (n)
RETURN n
ORDER BY n.name
LIMIT 3
```

结果

```
+-------------------+
| n                 |
+-------------------+
| Node[0]{name:"A"} |
| Node[1]{name:"B"} |
| Node[2]{name:"C"} |
+-------------------+
3 rows
```

3.3.16.2 返回来自表达式开始部分

LIMIT 接受来自任意表达式的正整数值，只要它不引用外部的变量。

参数

```
{{
"p" : 12
}
```

查询

```
MATCH (n)
RETURN n
ORDER BY n.name
LIMIT toInt(3 * rand())+ 1
```

结果

```
+-------------------+
| n                 |
+-------------------+
| Node[0]{name:"A"} |
| Node[1]{name:"B"} |
| Node[2]{name:"C"} |
+-------------------+
3 rows
```

3.3.17 SKIP

SKIP 定义了从哪行开始返回结果。

使用 SKIP 可以跳过开始的一部分结果。SKIP 图例如图 3-20 所示。

仅供非商业用途或交流学习使用

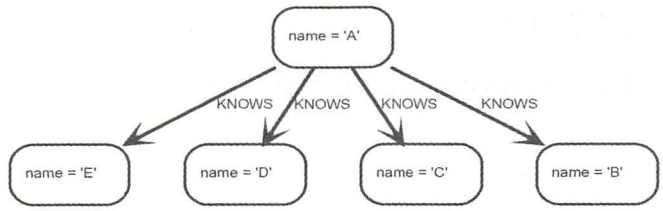


图 3-20　SKIP 图例

3.3.17.1　跳过前三

从第 4 个开始返回结果的一个子集，如下所示：

查询

```
MATCH (n)
RETURN n
ORDER BY n.name
SKIP 3
```

前三个节点被跳过了，只返回了最后两个节点。

结果

```
+-------------------+
| n                 |
+-------------------+
| Node[3]{name:"D"} |
| Node[4]{name:"E"} |
+-------------------+
2 rows
```

3.3.17.2　返回中间两个

从中间某个位置开始返回结果的字节，如下所示：

查询

```
MATCH (n)
RETURN n
ORDER BY n.name
SKIP 1
LIMIT 2
```

返回了中间的两个节点。

结果

```
+-------------------+
| n                 |
+-------------------+
| Node[1]{name:"B"} |
| Node[2]{name:"C"} |
+-------------------+
2 rows
```


仅供非商业用途或交流学习使用

3.3.17.3 跳过表达式的值加 1

SKIP 接受任意结果为正整数的值，只要它不引用其他外部变量。

查询

```
MATCH (n)
RETURN n
ORDER BY n.name
SKIP toInt(3*rand())+ 1
```

跳过前面三个节点，结果仅返回最后两个节点。

结果

```
+-------------------+
| n                 |
+-------------------+
| Node[3]{name:"D"} |
| Node[4]{name:"E"} |
+-------------------+
2 rows
```

3.3.18 WITH

WITH 语句将分段的查询部分连接在一起，查询结果从一部分以管道形式传递给另外一部分作为开始点。

使用 WITH 可以在将结果传递到后续查询之前对结果进行操作。操作可以是改变结果的形式或者数量。WITH 的一个常见用法就是限制传递给其他 MATCH 语句的结果数。通过结合 ORDER BY 和 LIMIT，可获取排在前面的 X 个结果。

另一个用法就是在聚合值上过滤。WITH 用于在 WHERE 断言中引入聚合。这些聚合表达式创建了新的结果绑定字段。WITH 也能像 RETURN 一样对结果使用别名作为绑定名。WITH 还可以用于将图的读语句和更新语句分开，查询中的每一部分要么只是读取，要么都是写入。当写部分的语句是基于读语句的结果时，这两者之间的转换必须使用 WITH。

WITH 图例如图 3-21 所示。

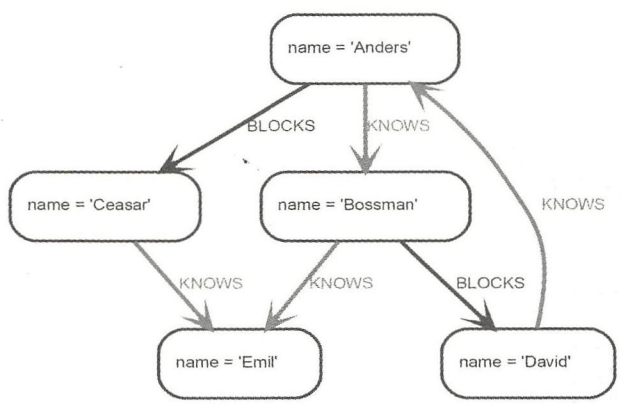


图 3-21 WITH 图例

仅供非商业用途或交流学习使用

3.3.18.1　过滤聚合函数结果

聚合的结果必须要通过 WITH 语句传递才能进行过滤。

查询

```
MATCH (david { name: 'David' })--(otherPerson)-->()
WITH otherPerson, count(*) AS foaf
WHERE foaf > 1
RETURN otherPerson
```

查询返回与'David'相连的人，且该人至少有一个外向关系。

结果

```
+-------------------------+
| otherPerson             |
+-------------------------+
| Node[0]{name:"Anders"} |
+-------------------------+
1 row
```

3.3.18.2　在 collect 前对结果排序

可以在将结果传递给 collect 函数之前对结果进行排序，这样就可以返回排过序的列表。

查询

```
MATCH (n)
WITH n
ORDER BY n.name DESC LIMIT 3
RETURN collect(n.name)
```

列表中的人名以倒序排列，并且数量限制为 3。

结果

```
+---------------------------+
| collect(n.name)           |
+---------------------------+
| ["Emil","David","Ceasar"] |
+---------------------------+
1 row
```

3.3.18.3　限制路径搜索的分支

可以限制匹配路径的数量，然后以这些路径为基础再做任何类似的有限制条件的搜索。

查询

```
MATCH (n { name: 'Anders' })--(m)
WITH m
ORDER BY m.name DESC LIMIT 1
MATCH (m)--(o)
RETURN o.name
```


仅供非商业用途或交流学习使用

先以“Anders”开始找到与之相连的所有节点，然后按名字倒序排列得到排第一个的节点。然后以该节点为基础，找到与之相连的所有节点，最后返回他们的名字。

结果

```
+-----------+
| o.name    |
+-----------+
| "Bossman" |
| "Anders"  |
+-----------+
2 rows
```

3.3.19 UNWIND

UNWIND 将一个列表展开为一个行的序列。

用 UNWIND 可以将任何列表转为单独的行。这些列表可以参数的形式传入，如前面 collect 函数的结果或者其他列表表达式。

UNWIND 一个较为常见的用法是创建唯一列表。另外一个是从提供给查询的参数列表中创建数据。UNWIND 需要给内部值指定新的名字。

3.3.19.1 UNIND 列表

将一个常量列表转为名为 x 的行并返回。

查询

```
UNWIND [1, 2, 3] AS x
RETURN x
```

原列表中的每个值以单独的行返回。

结果

```
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
+---+
3 rows
```

3.3.19.2 创建唯一列表

使用 DISTINCT 将一个重复值列表转为一个集合。

查询

```
WITH [1, 1, 2, 2] AS coll
UNWIND coll AS x
WITH DISTINCT x
RETURN collect(x) AS SET
```


仅供非商业用途或交流学习使用

原列表中的每个值被展开，然后经过 DISTINCT 之后创建了一个唯一列表。

结果

```
+-------+
| set   |
+-------+
| [1,2] |
+-------+
1 row
```

3.3.19.3　从列表参数创建节点

不用 FOREACH，通过列表参数来创建一系列节点和关系。

参数

```
{
"events" : [ {
"year" : 2014,
"id" : 1
}, {
"year" : 2014,
"id" : 2
} ]
}
```

查询

```
UNWIND $events AS event
MERGE (y:Year { year: event.year })
MERGE (y)<-[:IN]-(e:Event { id: event.id })
RETURN e.id AS x
ORDER BY x
```

原列表中的值被展开，通过 MERGE 来找到或者创建节点和关系。

结果

```
+---+
| x |
+---+
| 1 |
| 2 |
+---+
2 rows
Nodes created: 3
Relationships created: 2
Properties set: 3
Labels added: 3
```

3.3.20　UNION

UNION 语句用于将多个查询结果组合起来。

使用 UNION 组合查询的结果时，所有查询到的列的名称和数量必须完全一致。使用

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

UNION ALL 会包含所有结果行，而用 UNION 组合时，会移除结果集中的重复行。

UNION 图例如图 3-22 所示。

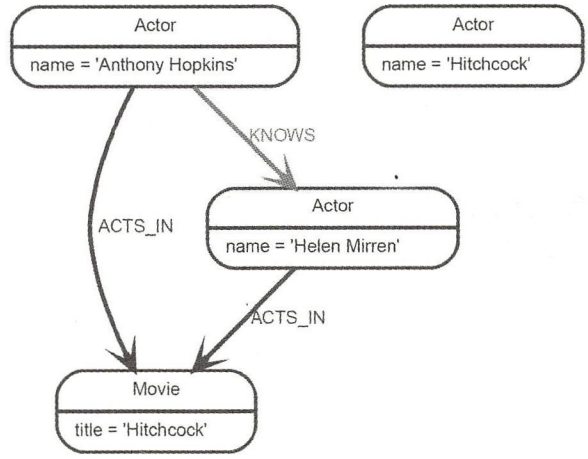


图 3-22　UNION 图例

3.3.20.1　组合两个查询

用 UNION ALL 将两个查询的结果组合在一起。

查询

```
MATCH (n:Actor)
RETURN n.name AS name
UNION ALL MATCH (n:Movie)
RETURN n.title AS name
```

返回了组合的结果，包含重复行。

结果

```
+--------------------+
| name               |
+--------------------+
| "Anthony Hopkins"  |
| "Helen Mirren"     |
| "Hitchcock"        |
| "Hitchcock"        |
+--------------------+
4 rows
```

3.3.20.2　组合两个查询并移除重复值

在 UNION 中不使用 ALL 时，组合的结果集中会去掉重复值。

查询

```
MATCH (n:Actor)
RETURN n.name AS name
```

仅供非商业用途或交流学习使用


```
UNION
MATCH (n:Movie)
RETURN n.title AS name
```

返回了没有重复值的组合结果。

结果

```
+-------------------+
| name              |
+-------------------+
| "Anthony Hopkins" |
| "Helen Mirren"    |
| "Hitchcock"       |
+-------------------+
3 rows
```

3.3.21　CALL

CALL 语句用于调用数据库中的过程（Procedure）。

使用 CALL 语句调用过程。调用过程时，需要指定所需要的参数。可以通过在过程名的后面使用逗号分隔的列表来显式地指定，也可以使用查询参数来作为过程调用的实参。后者仅适用于在单独的过程调用中作为参数，即整个查询语句只包含一个单一的 CALL 调用。

大多数的过程返回固定列的记录流，类似于 Cypher 查询返回的记录流。YIELD 子句用于显式地选择返回结果集中的哪些部分并绑定到一个变量以供后续的查询引用。在一个更大的查询内部，过程调用返回的结果可以显式地使用 YIELD 引入一个 WHERE 子句来过滤结果（类似 WITH ... WHERE ...）。

Neo4j 支持 VOID 过程。VOID 过程既没有声明任何结果字段，也不返回任何结果记录。调用 VOID 过程可能有一个副作用，就是它既不允许也不需要使用 YIELD。在一个大的查询中调用 VOID 过程，就像 WITH *在记录流的作用那样简单地传递输入的每一个结果。

下面的例子显示了如何传递实参并从一个过程中返回结果字段，所有的例子将使用下面的过程：

```
public class IndexingProcedure
{
@Context
public GraphDatabaseService db;
/**
* Adds a node to a named legacy index. Useful to, for instance, update
* a full-text index through cypher.
* @param indexName the name of the index in question
* @param nodeId id of the node to add to the index
* @param propKey property to index (value is read from the node)
*/
@Procedure(mode = Mode.WRITE)
public void addNodeToIndex( @Name("indexName") String indexName,
@Name("node") long nodeId,
@Name( value = "propKey", defaultValue = "name" ) String propKey )
{
```

仅供非商业用途或交流学习使用


```
Node node = db.getNodeById( nodeId );
db.index()
.forNodes( indexName )
.add( node, propKey, node.getProperty( propKey ) );
}
}
```

3.3.21.1 调用过程

本例调用数据库内嵌的过程 db.labels，它可列出数据库中的所有标签。

查询

```
CALL db.labels
```

结果

```
+-----------------+
| label           |
+-----------------+
| "User"          |
| "Administrator" |
+-----------------+
2 rows
```

3.3.21.2 使用命名空间和名字调用过程

本例调用数据库内嵌的过程 db.labels，它可列出数据库中的所有标签。

查询

```
CALL `db`.`labels`
```

结果

```
+-----------------+
| label           |
+-----------------+
| "User"          |
| "Administrator" |
+-----------------+
2 rows
```

3.3.21.3 使用字面值参数调用过程

下面使用字面值参数调用了例子中的过程 org.neo4j.procedure.example.addNodeToIndex，如参数直接写在语句中。

查询

```
CALL org.neo4j.procedure.example.addNodeToIndex('users', 0, 'name')
```

因为例子中的过程不返回任何结果，因此结果返回空。

仅供非商业用途或交流学习使用

结果

```
+-------------------------------------------+
| No data returned, and nothing was changed. |
+-------------------------------------------+
```

3.3.21.4　使用参数作为实参调用过程

这里使用参数作为实参调用了例子中的过程 org.neo4j.procedure.example.addNodeToIndex。每个过程的实参所取的值为参数语句中同名的参数对应的值（如果没有这个参数，则值为 null）。

参数

```
{
"indexName" : "users",
"node" : 0,
"propKey" : "name"
}
```

查询

```
CALL org.neo4j.procedure.example.addNodeToIndex
```

因为例子中的过程不返回任何结果，因此结果为空。

结果

```
+-------------------------------------------+
| No data returned, and nothing was changed. |
+-------------------------------------------+
```

3.3.21.5　混合使用字面值和参数调用过程

这里，混合使用字面值和参数来作为过程的参数来调用例子中的过程 org.neo4j.procedure.example.addNodeToIndex。

参数

```
{
"node" : 0
}
```

查询

```
CALL org.neo4j.procedure.example.addNodeToIndex('users', $node, 'name')
```

因为例子中的过程不返回任何结果，因此结果为空。

结果

```
+-------------------------------------------+
| No data returned, and nothing was changed. |
+-------------------------------------------+
```

仅供非商业用途或交流学习使用

3.3.21.6 使用字面值和默认实参调用过程

这里使用字面值和过程本身提供的默认值作为参数，调用例子中的过程org.neo4j.procedure.example.addNodeToIndex。

查询

```
CALL org.neo4j.procedure.example.addNodeToIndex('users', 0)
```

因为例子中的过程不返回任何结果，因此结果为空。

结果

```
+-------------------------------------------+
| No data returned, and nothing was changed. |
+-------------------------------------------+
```

3.3.21.7 在复杂查询中调用过程

这里调用数据库内嵌的过程 db.labels 计算数据库中的总标签数。

查询

```
CALL db.labels() YIELD label
RETURN count(label) AS numLabels
```

因为过程调用是大的查询的一部分，所以所有的输出都必须显式地命名。

结果

```
+-----------+
| numLabels |
+-----------+
| 2         |
+-----------+
1 row
```

3.3.21.8 调用过程并过滤结果

这里调用数据库内嵌的过程 db.labels，并计算数据库中所有在用的包含“User”词的标签。

查询

```
CALL db.labels() YIELD label
WHERE label CONTAINS 'User'
RETURN count(label) AS numLabels
```

因为过程调用是大的查询的一部分，所以所有的输出都必须显式地命名。

结果

```
+-----------+
| numLabels |
+-----------+
| 1         |
+-----------+
1 row
```

仅供非商业用途或交流学习使用

3.3.21.9　在复杂查询中调用过程并重命名结果

这里调用内嵌过程 db.propertyKeys 作为一部分，计算数据库中包含每个属性键的节点数。

查询

```
CALL db.propertyKeys() YIELD propertyKey AS prop
MATCH (n)
WHERE n[prop] IS NOT NULL RETURN prop, count(n) AS numNodes
```

因为过程调用是大的查询的一部分，所以所有的输出都必须显式地命名。

结果

```
+--------+----------+
| prop   | numNodes |
+--------+----------+
| "name" | 1        |
+--------+----------+
1 row
```

3.4 函数

本节包含了 Cypher 查询语言中的所有函数的信息。

3.4.1　断言（Predicate）函数

断言函数是对给定的输入返回 true 或者 false 的布尔函数，它们主要用于查询的 WHERE 部分过滤子图。Predicate 图例如图 3-23 所示。

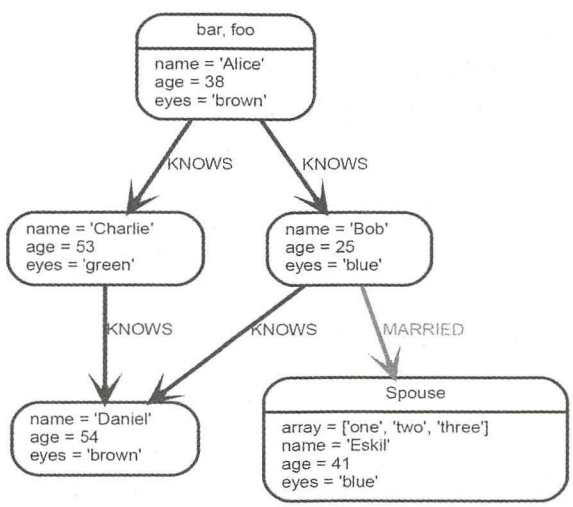


图 3-23　Predicate 图例

仅供非商业用途或交流学习使用

3.4.1.1 all()

判断是否一个断言适用于列表中的所有元素。

语法：all(variable IN list WHERE predicate)

参数：

- list: 返回列表的表达式。
- variable: 用于断言中的变量。
- predicate: 用于测试列表中所有元素的断言。

查询

```
MATCH p =(a)-[*1..3]->(b)
WHERE a.name = 'Alice' AND b.name = 'Daniel' AND ALL (x IN nodes(p) WHERE
x.age > 30)
RETURN p
```

返回路径中的所有节点都有一个至少大于 30 的 age 属性。

结果

```
+--------------------------------------------------------------------------------+
| p                                                                              |
+--------------------------------------------------------------------------------+
| [Node[0]{name:"Alice",age:38,eyes:"brown"},:KNOWS[1]{},Node[2]{name:"Charlie", |
|  age:53,eyes:"green"},:KNOWS[3]{},Node[3]{name:"Daniel",age:54,eyes:"brown"}]  |
+--------------------------------------------------------------------------------+
1 row
```

3.4.1.2 any()

判断是否一个断言至少适用于列表中的一个元素。

语法：any(variable IN list WHERE predicate)

参数：

- list: 返回列表的表达式。
- variable: 用于断言中的变量。
- predicate: 用于测试列表中所有元素的断言。

查询

```
MATCH (a)
WHERE a.name = 'Eskil' AND ANY (x IN a.array WHERE x = 'one')
RETURN a
```

返回路径中的所有节点的 array 数组属性中至少有一个值为'one'。

结果

```
+-----------------------------------------------------------------------+
| a                                                                     |
+-----------------------------------------------------------------------+
| Node[4]{array:["one","two","three"],name:"Eskil",age:41,eyes:"blue"} |
+-----------------------------------------------------------------------+
1 row
```



仅供非商业用途或交流学习使用

3.4.1.3　none()

如果断言不适用于列表中的任何元素，则返回 true。

语法：none(variable IN list WHERE predicate)
参数：

- list: 返回列表的表达式。
- variable: 用于断言中的变量。
- predicate: 用于测试列表中所有元素的断言。

查询

```
MATCH p =(n)-[*1..3]->(b)
WHERE n.name = 'Alice' AND NONE (x IN nodes(p) WHERE x.age = 25)
RETURN p
```

返回的路径中没有节点的 age 属性值为 25。

结果

```
+--------------------------------------------------------------------------------+
| p                                                                              |
+--------------------------------------------------------------------------------+
[ Node[0]{name:"Alice",age:38,eyes:"brown"},:KNOWS[1]{},Node[2]{name:"Charlie",  |
|   age:53,eyes:"green"}]                                                        |
[ Node[0]{name:"Alice",age:38,eyes:"brown"},:KNOWS[1]{},Node[2]{name:"Charlie",  |
|   age:53,eyes:"green"},:KNOWS[3]{},Node[3]{name:"Daniel",age:54,eyes:"brown"}] |
+--------------------------------------------------------------------------------+
2 rows
```

3.4.1.4　single()

如果断言刚好只适用于列表中的某一个元素，则返回 true。

语法：single(variable IN list WHERE predicate)
参数：

- list: 返回列表的表达式。
- variable: 用于断言中的变量。
- predicate: 用于测试列表中所有元素的断言。

查询

```
MATCH p =(n)-->(b)
WHERE n.name = 'Alice' AND SINGLE (var IN nodes(p) WHERE var.eyes = 'blue')
RETURN p
```

每条返回的路径中刚好只有一个节点的 eyes 属性值为'blue'。



结果

```
+--------------------------------------------------------------------------------+
| p                                                                              |
+--------------------------------------------------------------------------------+
| [Node[0]{name:"Alice",age:38,eyes:"brown"},:KNOWS[0]{},Node[1]{name:"Bob",      |
|  age:25,eyes:"blue"}]                                                          |
+--------------------------------------------------------------------------------+
1 row
```

3.4.1.5 exists()

如果数据库中存在该模式或者节点中存在该属性时，则返回 true。

语法：exists(pattern-or-property)

参数：

- pattern-or-property：模式或者属性（以“variable.prop”的形式）。

查询

```
M MATCH (n)
WHERE exists(n.name)
RETURN n.name AS name, exists((n)-[:MARRIED]->()) AS is_married
```

本查询返回了所有节点的 name 属性和一个指示是否已婚的 true/false 值。

结果

```
+-----------+------------+
| name      | is_married |
+-----------+------------+
| "Alice"   | false      |
| "Bob"     | true       |
| "Charlie" | false      |
| "Daniel"  | false      |
| "Eskil"   | false      |
+-----------+------------+
5 rows
```

3.4.2 标量（Scalar）函数

标量函数返回一个单值。Scalar 图例如图 3-24 所示。



length()和 size()函数非常相似，因此弄清它们的区别就显得很重要。为了保持向后兼容，length()目前只适用于 4 种类型：字符串、路径、列表和模式表达式。然而，为了清晰起见，推荐仅仅在字符串和路径上使用 length()，在列表和模式表达式上使用新的 size()函数。length()在这些类型上的功能在将来可能被弃用。



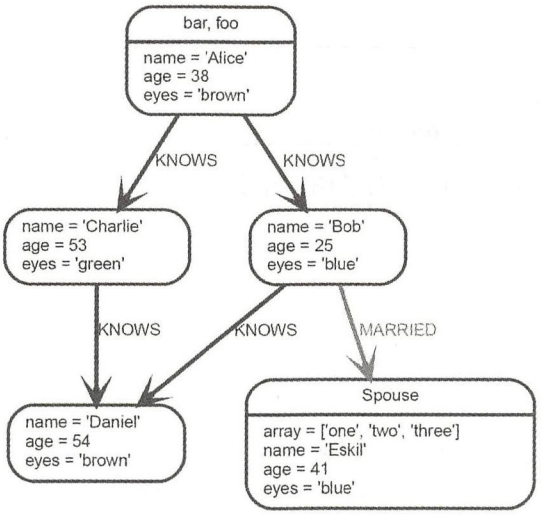


图 3-24　Scalar 图例

3.4.2.1　size()

使用 size()返回表中元素的个数。

语法：size(list)

参数：

- list: 返回列表的表达式。

查询

```
RETURN size(['Alice', 'Bob']) AS col
```

本查询返回了列表中元素的个数。

结果

```
+-----+
| col |
+-----+
| 2   |
+-----+
1 row
```

3.4.2.2　模式表达式的 size

这里的 size()的参数不是一个列表，而是一个模式表达式匹配到的查询结果集。计算的是结果集元素的个数，而不是表达式本身的长度。

语法：size(pattern expression)

参数：

- pattern expression: 返回列表的模式表达式。



查询

```
MATCH (a)
WHERE a.name = 'Alice'
RETURN size((a)-->()-->()) AS fof
```

本查询返回了模式表达式匹配到的子图的个数。

结果

```
+-----+
| fof |
+-----+
| 3   |
+-----+
1 row
```

3.4.2.3 length()

使用 length()函数返回路径的长度。

语法：length(path)

参数：

- path: 返回路径的表达式。

查询

```
MATCH p =(a)-->(b)-->(c)
WHERE a.name = 'Alice'
RETURN length(p)
```

本查询返回路径 p 的长度。

结果

```
+-----------+
| length(p) |
+-----------+
| 2         |
| 2         |
| 2         |
+-----------+
3 rows
```

3.4.2.4 字符串的长度

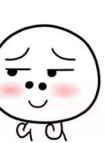
语法：length(string)

参数：

- string: 返回字符串的表达式。

查询

```
MATCH (a)
WHERE length(a.name)> 6
RETURN length(a.name)
```

查询返回了 name 为'Charlie'的长度。

结果

```
+-----------------+
| length(a.name) |
+-----------------+
| 7              |
+-----------------+
1 row
```

3.4.2.5　type()

返回字符串代表的关系类型。

语法：type(relationship)

参数：

- relationship: 一个关系。

查询

```
MATCH (n)-[r]->()
WHERE n.name = 'Alice'
RETURN type(r)
```

查询返回了关系 r 的关系类型。

结果

```
+---------+
| type(r) |
+---------+
| "KNOWS" |
| "KNOWS" |
+---------+
2 rows
```

3.4.2.6　id()

返回关系或者节点的 id。

语法：id(property-container)

参数：

- property-container: 一个节点或者关系。

查询

```
MATCH (a)
RETURN id(a)
```

返回了 5 个节点的 id。

结果

```
+-------+
| id(a) |
+-------+
```



```
| 0     |
| 1     |
| 2     |
| 3     |
| 4     |
+-------+
5 rows
```

3.4.2.7 coalesce()

返回表达式列表中的第一个非空的值。如果所有的实参都为空 ，则返回 null。

语法：coalesce(expression [, expression]*)

参数：

- expression: 表达式，可能返回 null。

查询

```
MATCH (a)
WHERE a.name = 'Alice'
RETURN coalesce(a.hairColor, a.eyes)
```

结果

```
+------------------------------+
| coalesce(a.hairColor, a.eyes) |
+------------------------------+
| "brown"                      |
+------------------------------+
1 row
```

3.4.2.8 head()

head()返回列表中的第一个元素。

语法：head(expression)

参数：

- expression: 返回列表的表达式。

查询

```
MATCH (a)
WHERE a.name = 'Eskil'
RETURN a.array, head(a.array)
```

返回了路径中的第一个节点。

结果

```
+----------------------+---------------+
| a.array              | head(a.array) |
+----------------------+---------------+
| ["one","two","three"] | "one"         |
+----------------------+---------------+
1 row
```



3.4.2.9　last()

last()返回列表中的最后一个元素。

语法：last(expression)

参数：

- expression: 返回列表的表达式。

查询

```
MATCH (a)
WHERE a.name = 'Eskil'
RETURN a.array, last(a.array)
```

返回了路径中的最后一个节点。

结果

```
+----------------------+---------------+
| a.array              | last(a.array) |
+----------------------+---------------+
| ["one","two","three"] | "three"       |
+----------------------+---------------+
1 row
```

3.4.2.10　timestamp()

timestamp()返回当前时间与 1970 年 1 月 1 日午夜之间的差值，单位以毫秒计算。它在整个查询中始终返回同一个值，即使是在一个运行时间很长的查询中。

语法：timestamp()

参数：无

查询

```
RETURN timestamp()
```

以毫秒返回当前时间。

结果

```
+---------------+
| timestamp()   |
+---------------+
| 1486061001197 |
+---------------+
1 row
```

3.4.2.11　startNode()

startNode()返回一个关系的开始节点。

语法：startNode(relationship)

参数：

- relationship: 返回关系的表达式。



查询

```
MATCH (x:foo)-[r]-()
RETURN startNode(r)
```

结果

```
+-----------------------------------------+
| startNode(r)                            |
+-----------------------------------------+
| Node[0]{name:"Alice",age:38,eyes:"brown"} |
| Node[0]{name:"Alice",age:38,eyes:"brown"} |
+-----------------------------------------+
2 rows
```

3.4.2.12 endNode()

endNode()返回一个关系的结束节点。

语法：endNode(relationship)

参数：

- relationship: 返回关系的表达式。

查询

```
MATCH (x:foo)-[r]-()
RETURN endNode(r)
```

结果

```
+-----------------------------------------+
| endNode(r)                              |
+-----------------------------------------+
| Node[2]{name:"Charlie",age:53,eyes:"green"} |
| Node[1]{name:"Bob",age:25,eyes:"blue"}      |
+-----------------------------------------+
2 rows
```

3.4.2.13 properties()

properties()将实参转为属性值的 map。如果实参是一个节点或者关系，返回的就是节点或关系的属性的 map；如果实参已经是一个 map 了，那么原样返回结果。

语法：properties(expression)

参数：

- expression: 返回节点、关系或者 map 的表达式。

查询

```
C CREATE (p:Person { name: 'Stefan', city: 'Berlin' })
RETURN properties(p)
```

结果

```
+-----------------------------------+
| properties(p)                     |
+-----------------------------------+
```

```
| {name -> "Stefan", city -> "Berlin"} |
+--------------------------------------+
1 row
Nodes created: 1
Properties set: 2
Labels added: 1
```

3.4.2.14　toInt()

toInt()将实参转换为一个整数。字符串会被解析为一个整数，如果解析失败，将返回null。浮点数将被强制转换为整数。

语法：toInt(expression)

参数：

- expression: 返回任意值的表达式。

查询

```
R RETURN toInt('42'), toInt('not a number')
```

结果

```
+-------------+-----------------------+
| toInt('42') | toInt('not a number') |
+-------------+-----------------------+
| 42          | <null>                |
+-------------+-----------------------+
1 row
```

3.4.2.15　toFloat

toFloat()将实参转换为浮点数。字符串会被解析为一个浮点数，如果解析失败，将返回null。整数将被强制转换为浮点数。

语法：toFloat(expression)

参数：

- expression: 返回任意值的表达式。

查询

```
RETURN toFloat('11.5'), toFloat('not a number')
```

结果

```
+-----------------+-------------------------+
| toFloat('11.5') | toFloat('not a number') |
+-----------------+-------------------------+
| 11.5            | <null>                  |
+-----------------+-------------------------+
1 row
```

3.4.3　列表（List）函数

列表函数返回列表中的元素，如路径中的节点等。

List 图例如图 3-25 所示。

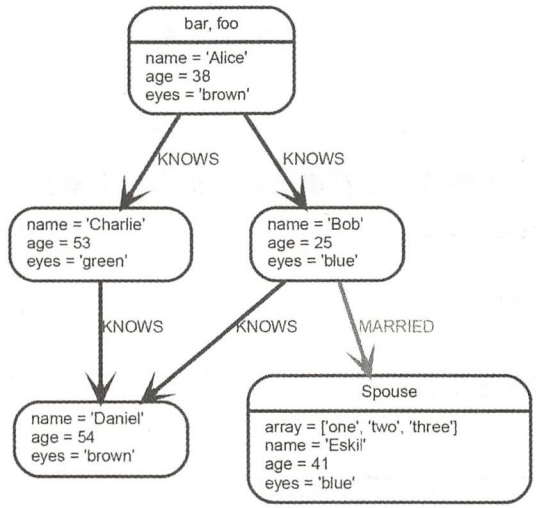


图 3-25　List 图例

3.4.3.1　nodes()

返回一条路径中的所有节点。

语法：nodes(path)

参数：

- path：一条路径。

查询返回了路径 p 中的所有节点。

查询

```
MATCH p =(a)-->(b)-->(c)
WHERE a.name = 'Alice' AND c.name = 'Eskil'
RETURN nodes(p)
```

结果

```
+--------------------------------------------------------------------------------+
| nodes(p)                                                                       |
+--------------------------------------------------------------------------------+
|[Node[0]{name:"Alice",age:38,eyes:"brown"},Node[1]{name:"Bob",age:25,yes:"blue"},|
|   Node[4]{array:["one","two","three"],name:"Eskil",age:41,eyes:"blue"}]         |
+--------------------------------------------------------------------------------+
1 row
```

3.4.3.2　relationships()

返回一条路径中的所有关系。

语法：relationships(path)

参数：

- path: 一条路径。

查询

```
MATCH p =(a)-->(b)-->(c)
WHERE a.name = 'Alice' AND c.name = 'Eskil'
RETURN relationships(p)
```

查询返回了路径 p 中的所有节点关系。

结果

```
+----------------------------+
| relationships(p)           |
+----------------------------+
| [:KNOWS[0]{},:MARRIED[4]{}] |
+----------------------------+
1 row
```

3.4.3.3 labels()

以字符串列表的形式返回一个节点的所有标签。

语法：labels(node)

参数：

- node: 返回单个节点的任意表达式。

查询

```
MATCH (a)
WHERE a.name = 'Alice'
RETURN labels(a)
```

查询返回了节点 n 的所有标签。

结果

```
+---------------+
| labels(a)     |
+---------------+
| ["bar","foo"] |
+---------------+
1 row
```

3.4.3.4 keys()

以字符串列表的形式返回一个节点、关系或者 map 的所有属性的名称。

语法：keys(property-container)

参数：

- property-container: 一个节点、关系或者字面值的 map。

查询

```
MATCH (a)
WHERE a.name = 'Alice'
```

```
RETURN keys(a)
```

查询返回了节点 a 的属性名。

结果

```
+----------------------+
| keys(a)              |
+----------------------+
| ["name","age","eyes"] |
+----------------------+
1 row
```

3.4.3.5 extract()

可以使用 extract()从节点或关系列表中返回单个属性或者某个函数的值。它将遍历整个列表，针对列表中的每个元素运行一个表达式，然后以列表的形式返回这些结果。它的工作方式类似于 Lisp 和 Scala 等函数式语言中的 map 方法。

语法：extract(variable IN list | expression)

参数：

- list: 返回列表的表达式。
- variable: 引用 list 中元素的变量，它在 expression 中会用到。
- expression: 针对列表中每个元素所运行的表达式，并产生一个结果列表。

查询

```
MATCH p =(a)-->(b)-->(c)
WHERE a.name = 'Alice' AND b.name = 'Bob' AND c.name = 'Daniel'
RETURN extract(n IN nodes(p)| n.age) AS extracted
```

返回了路径中所有节点的 age 属性。

结果

```
+------------+
| extracted  |
+------------+
| [38,25,54] |
+------------+
1 row
```

3.4.3.6 filter()

filter()返回列表中满足断言要求的所有元素。

语法：filter(variable IN list WHERE predicate)

参数：

- list: 返回列表的表达式。
- variable: 断言中引用列表元素所用到的变量。
- predicate: 针对列表中每个元素进行测试的断言。

查询

```
MATCH (a)
WHERE a.name = 'Eskil'
RETURN a.array, filter(x IN a.array WHERE size(x)= 3)
```

返回 array 属性，及其元素的字符数为 3 的元素列表。

结果

```
+----------------------+----------------------------------------+
| a.array              | filter(x IN a.array WHERE size(x) = 3) |
+----------------------+----------------------------------------+
| ["one","two","three"] | ["one","two"]                          |
+----------------------+----------------------------------------+
1 row
```

3.4.3.7　tail()

tail()返回列表中除了首元素之外的所有元素。

语法：tail(expression)

参数：

- expression: 返回某个类型列表的表达式。

查询

```
MATCH (a)
WHERE a.name = 'Eskil'
RETURN a.array, tail(a.array)
```

返回了 array 属性及属性中除了第一个之外的所有元素。

结果

```
+----------------------+-----------------+
| a.array              | tail(a.array)   |
+----------------------+-----------------+
| ["one","two","three"] | ["two","three"] |
+----------------------+-----------------+
1 row
```

3.4.3.8　range()

range()返回某个范围内的数值。值之间的默认步长为 1，范围包含起始边界值。

语法：range(start, end [, step])

参数：

- start: 起点数值的表达式。
- end: 结束数值的表达式。
- step: 数值间隔的步长。

查询

```
RETURN range(0, 10), range(2, 18, 3)
```

第一个返回了 0 到 10 之间步长为 1 的所有值，第二个返回了从 2 到 18 之间步长为 3 的所有值。

结果

```
+--------------------------+------------------+
| range(0, 10)             | range(2, 18, 3)  |
+--------------------------+------------------+
| [0,1,2,3,4,5,6,7,8,9,10] | [2,5,8,11,14,17] |
+--------------------------+------------------+
1 row
```

3.4.3.9 reduce()

可以用 reduce()对列表中的每个元素执行一个表达式，将表达式结果存入一个累加器。它的工作机制类似 Lisp 和 Scala 等函数式语言中的 fold 或者 reduce 方法。

语法：reduce(accumulator = initial, variable IN list | expression)

参数：

- accmulator: 用于累加每次迭代的部分结果。
- initial: 累加器的初始值。
- list: 列表。
- variable: 引用列表中的每个元素的变量。
- expression: 针对列表中每个元素执行的表达式。

查询

```
MATCH p =(a)-->(b)-->(c)
WHERE a.name = 'Alice' AND b.name = 'Bob' AND c.name = 'Daniel'
RETURN reduce(totalAge = 0, n IN nodes(p)| totalAge + n.age) AS reduction
```

本查询将路径中每个节点的 age 数值加起来，然后返回一个单值。

结果

```
+-----------+
| reduction |
+-----------+
| 117       |
+-----------+
1 row
```

3.4.4 数学函数

这些函数仅适用于数值表达式。如果用于其他类型的值，将返回错误。

数学函数图例如图 3-26 所示。

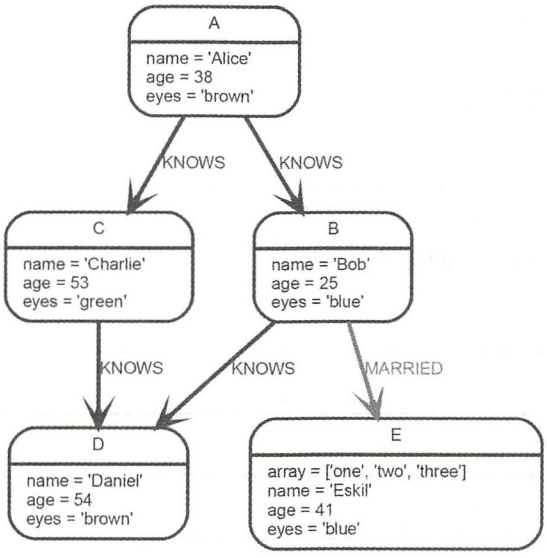


图 3-26 数学函数图例

1. 数值函数

（1）abs()

abs()函数返回数值的绝对值。

语法：abs(expression)

参数：

- expression: 数值表达式。

查询

```
MATCH (a),(e)
WHERE a.name = 'Alice' AND e.name = 'Eskil'
RETURN a.age, e.age, abs(a.age - e.age)
```

返回两个年龄差值的绝对值。

结果

a.age	e.age	abs(a.age - e.age)
38	41	3
1 row		

（2）ceil()

ceil()返回大于或等于实参的最小整数。

语法：ceil(expression)
参数：

- expression：数值表达式。

查询

```
RETURN ceil(0.1)
```

返回大于 0.1 的最小整数，即为 1。

结果

ceil(0.1)
1.0
1 row

（3）floor()
floor()返回小于等于表达式的最大的整数。

语法：floor(expression)
参数：

- expression：数值表达式。

查询

RETURN floor(0.9)
小于等于 0.9 的最大的整数为 0。

结果

floor(0.9)
0
1 row

（4）round()
round()返回距离表达式值最近的整数。

语法：round(expression)
参数：

- expression：数值表达式。

查询

```
RETURN round(3.141592)
```

距离 3.141592 最近的整数为 3。

结果

round(3.141592)
3
1 row

（5）sign()

sign()返回一个数值的正负。如果值为零，则返回 0；如果值为负数，则返回-1；如果值为正数，返回 1。

语法：sign(expression)

参数：

- expression：数值表达。

查询

```
RETURN sign(-17), sign(0.1)
```

返回了-17 和 0.1 的正负符号。

结果

sign(-17)	sign(0.1)
-1	1
1 row	

（6）rand()

rand()返回[0, 1)之间的一个随机数，返回的数值在整个区间遵循均匀分布。

语法：rand()

参数：无

查询

```
RETURN rand()
```

返回了一个随机数。

结果

rand()
0.6835626594404169

1 row

2. 对数函数

（1）log()

log()返回表达式的自然对数。

语法：log(expression)

参数：

- expression：数值表达式。

查询

RETURN log(27)

返回了 27 的自然对数的值。

结果

log(27)
3.295836866004329
1 row

（2）log10()

log10()返回表达式的常用对数(以 10 为底)。

语法：log10(expression)

参数：

- expression：数值表达式。

查询

```
RETURN log10(27)
```

返回了 27 的常用对数。

结果

log10(27)
1.4313637641589874
1 row

（3）exp()

exp()返回 e^n，这里 e 是自然对数的底，n 是表达式的实参值。

语法：e(expression)
参数：

- expression: 数值表达式。

查询

```
RETURN exp(2)
```

返回了 e 的平方值。

结果

exp(2)
7.38905609893065
1 row

（4）e()

e()返回自然对数的底，即 e。

语法：e()
参数：无

查询

```
RETURN e()
```

返回了自然对数的底 e 的值。

结果

e()
2.718281828459045
1 row

（5）sqrt()

sqrt()返回数值的平方根。

语法：sqrt(expression)
参数：

- expression: 数值表达式。

查询

```
RETURN sqrt(256)
```

返回了 256 的平方根，即 16。

sqrt(256)
16
1 row

3. 三角函数

除非特别指明，所有的三角函数都是针对弧度值进行计算的。

（1）sin()

sin()返回表达式的正弦函数值。

语法：sin(expression)

参数：

- expression: 一个表示角的弧度的数值表达式。

查询

```
RETURN sin(0.5)
```

返回了弧度为 0.5 的正弦值。

结果

sin(0.5)
0.479425538604203
1 row

（2）cos()

cos()返回表达式的余弦函数值。

语法：cos(expression)

参数：

- expression: 一个表示角弧度的数值表达式。

查询

```
RETURN cos(0.5)
```

返回了弧度为 0.5 的余弦值。

结果

cos(0.5)
0.8775825618903728
1 row

（3）tan()

tan()返回表达式的正切值。

语法：tan(expression)

参数：

- expression: 一个表示角弧度的数值表达式。

查询

```
RETURN tan(0.5)
```

返回了弧度为 0.5 的正切值。

结果

tan(0.5)
0.5463024898437905
1 row

（4）cot()

cot()返回表达式的余切值。

语法：cot(expression)

参数：

- expression: 一个表示角弧度的数值表达式。

查询

```
RETURN cot(0.5)
```

返回了弧度为 0.5 的余切值。

结果

cot(0.5)
1.830487721712452
1 row

（5）asin()

asin()返回表达式的反正弦值。

语法：asin(expression)

参数：

- expression: 一个表示角弧度的数值表达式。

查询

```
RETURN asin(0.5)
```

返回了弧度为 0.5 的反正弦值。

结果

asin(0.5)
0.5235987755982989
1 row

（6）acos()

acos()返回表达式的反余弦值。

语法：acos(expression)

参数：

- expression: 一个表示角弧度的数值表达式。

查询

```
RETURN acos(0.5)
```

返回了弧度为 0.5 的反余弦值。

结果

acos(0.5)
1.0471975511965979
1 row

（7）atan()

atan()返回表达式的反正切值。

语法：atan(expression)

参数：

- expression: 一个表示角弧度的数值表达式。

查询

```
RETURN atan(0.5)
```

返回了弧度为 0.5 的反正切值。

结果

atan(0.5)
0.4636476090008061
1 row

（8）atan2()

atan2()返回方位角，也可以理解为计算复数 x+yi 的幅角。

语法：atan2(expression1, expression2)

参数：

- expression1：表示复数 x 部分的数值表达式。
- expression2：表示复数 y 部分的数值表达式。

查询

```
RETURN atan2(0.5, 0.6)
```

返回了复数为 0.5+0.6i 的方位角。

结果

atan2(0.5, 0.6)
0.6947382761967033
1 row

（9）pi()

pi()返回常数 pi 的数值。

语法：pi()

参数：无

查询

```
RETURN pi()
```

返回了常数Π的值。

结果

pi()
3.141592653589793
1 row

（10）degrees()

degrees()将弧度转为度。

语法：degrees(expression)

参数：

- expression: 一个表示角弧度的数值表达式。

查询

```
RETURN degrees(3.14159)
```

返回了接近于Π的度数。

结果

degrees(3.14159)
179.99984796050427
1 row

（11）radians()

radians()将度转为弧度。

语法：radians(expression)

参数：

- expression: 一个表示角度数的数值表达式。

查询

```
RETURN radians(180)
```

返回了 180 度的弧度值（Π）。

结果

radians(180)
3.141592653589793
1 row

（12）haversin()

haversin()返回表达式的半正矢。

语法：haversin(expression)

参数：

- expression: 一个表示角弧度的数值表达式。

查询

```
RETURN haversin(0.5)
```

返回了弧度为 0.5 的半正矢值。

结果

haversin(0.5)
0.06120871905481362
haversin(0.5)
1 row

（13）使用 haversin 函数计算球面距离

haversin()函数可用于计算球面上两点（以经纬度的方式给出）之间的距离。在本例计算了德国柏林（lat 52.5，lon 13.4）和美国加州圣马特奥市（lat 375，lon -122.3）两点之间的球面距离（以 km 计），计算时采用的是地球的平均半径 6371km。

查询

```
CREATE (ber:City { lat: 52.5, lon: 13.4 }),(sm:City { lat: 37.5, lon: -
122.3 })
RETURN 2 * 6371 * asin(sqrt(haversin(radians(sm.lat - ber.lat))+
cos(radians(sm.lat))*cos(radians(ber.lat))* haversin(radians(sm.lon -
ber.lon)))) AS dist
```

返回了柏林和圣马特奥之间的估算距离。

结果

Dist
9129.969740051658
1 row Nodes created: 2 Properties set: 4 Labels added: 2

3.4.5　字符串函数

下面的函数都是只针对字符串表达式。如果用于其他值，将返回错误。有个例外就是 toString()，它还接受数字值和布尔值。

字符串函数图例如图 3-27 所示。

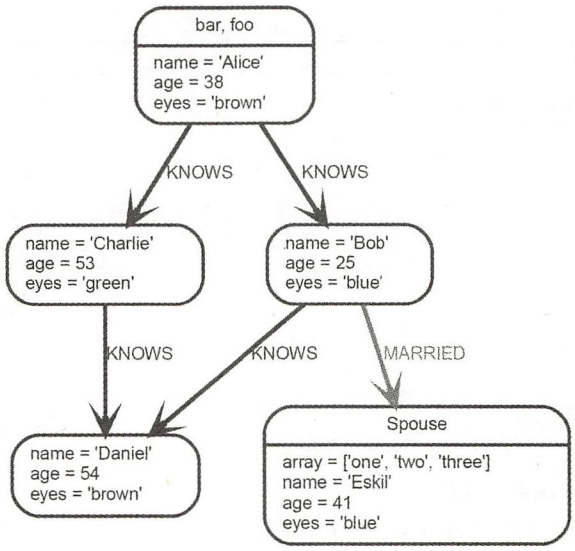


图 3-27　字符串函数图例

3.4.5.1　replace()

replace()返回被替换字符串替换后的字符串，它会替换所有出现过的字符串。

语法：replace(original, search, replace)

参数：

- original: 原字符串。
- search: 期望被替换的字符串。
- replace: 用于替换的字符串。

查询

```
RETURN replace("hello", "l", "w")
```

结果

```
+----------------------------+
| replace("hello", "l", "w") |
+----------------------------+
| "hewwo"                    |
+----------------------------+
1 row
```

3.4.5.2　substring()

substring()返回原字符串的子串。它带有一个 0 为开始的索引值和长度作为参数。如果长度省略了，那么它返回从索引开始到结束的子字符串。

语法：substring(original, start [, length])

参数：

- original: 原字符串。
- start: 子串的开始位置。
- length: 子串的长度。

查询

```
RETURN substring('hello', 1, 3), substring('hello', 2)
```

结果

```
+--------------------------+----------------------+
| substring('hello', 1, 3) | substring('hello', 2) |
+--------------------------+----------------------+
| "ell"                    | "llo"                 |
+--------------------------+----------------------+
1 row
```

3.4.5.3 left()

left()返回原字符串左边指定长度的子串。

语法：left(original, length)

参数：

- original: 原字符串。
- length: 左边子字符串的长度。

查询

```
RETURN left('hello', 3)
```

结果

```
+------------------+
| left('hello', 3) |
+------------------+
| "hel"            |
+------------------+
1 row
```

3.4.5.4 right()

right()返回原字符串右边的指定长度的子字符串。

语法：right(original, length)

参数：

- original: 原字符串。
- length: 右边子字符串的长度。

查询

```
RETURN right('hello', 3)
```

结果

```
+-------------------+
| right('hello', 3) |
+-------------------+
| "llo"             |
+-------------------+
1 row
```

3.4.5.5 ltrim()

ltrim()返回原字符串移除左侧的空白字符后的字符串。

语法：ltrim(original)

参数：

- original: 原字符串。

查询

```
RETURN ltrim(' hello')
```

结果

```
+-------------------+
| ltrim(' hello')   |
+-------------------+
| "hello"           |
+-------------------+
1 row
```

3.4.5.6 rtrim()

rtrim()返回原字符串移除右侧空白字符后的字符串。

语法：rtrim(original)

参数：

- original: 原字符串。

查询

```
RETURN rtrim('hello ')
```

结果

```
+-------------------+
| rtrim('hello ')   |
+-------------------+
| "hello"           |
+-------------------+
1 row
```

3.4.5.7 trim()

trim()返回原字符串移除两侧的空白字符之后的字符串。

语法：trim(original)

参数：

- original: 原字符串。

查询

```
RETURN trim(' hello ')
```

结果

```
+--------------------+
| trim(' hello ')    |
+--------------------+
| "hello"            |
+--------------------+
1 row
```

3.4.5.8　lower()

lower()以小写的形式返回原字符串。

语法：lower(original)

参数：

- original: 原字符串。

查询

```
RETURN lower('HELLO')
```

结果

```
+----------------+
| lower('HELLO') |
+----------------+
| "hello"        |
+----------------+
1 row
```

3.4.5.9　upper()

uppper()以大写的形式返回原字符串。

语法：upper(original)

参数：

- original: 原字符串。

查询

```
RETURN upper('hello')
```

结果

```
+----------------+
| upper('hello') |
+----------------+
| "HELLO"        |
+----------------+
1 row
```

3.4.5.10 split()

split()返回以指定模式分隔后的字符串序列。

语法：split(original, splitPattern)

参数：

- original: 原字符串。
- splitPattern: 分割字符串。

查询

```
RETURN split('one,two', ',')
```

结果

```
+-----------------------+
| split('one,two', ',') |
+-----------------------+
| ["one","two"]         |
+-----------------------+
1 row
```

3.4.5.11 reverse()

reverse()返回原字符串的倒序字符串。

语法：reverse(original)

参数：

- original: 原字符串。

查询

```
RETURN reverse('anagram')
```

结果

```
+--------------------+
| reverse('anagram') |
+--------------------+
| "margana"          |
+--------------------+
1 row
```

3.4.5.12 toString()

toString()将实参转换为字符串。它将整型、浮点型和布尔型转换为字符串。如果实参为字符串，则按原样返回。

语法：toString(expression)

参数：

- expression: 返回数值、布尔或者字符串的表达式。

查询

```
RETURN toString(11.5), toString('already a string'), toString(TRUE )
```

结果

```
+---------------+-----------------------------+----------------+
| toString(11.5) | toString('already a string') | toString(true) |
+---------------+-----------------------------+----------------+
| "11.5"         | "already a string"           | "true"         |
+---------------+-----------------------------+----------------+
1 row
```

3.4.6　自定义函数

自定义函数用 Java 语言编写，可部署到数据库中，调用方式与其他 Cypher 函数一样。下面的例子展示了如何调用一个名为 join 的自定义函数。

3.4.6.1　调用自定义函数

调用自定义函数 org.neo4j.procedure.example.join()。

查询

```
MATCH (n:Member)
RETURN org.neo4j.function.example.join(collect(n.name))
```

结果

```
+-----------------------------------------------+
| org.neo4j.function.example.join(collect(n.name)) |
+-----------------------------------------------+
| "John,Paul,George,Ringo"                      |
+-----------------------------------------------+
1 row
```

3.4.6.2　编写自定义函数

自定义函数的编写类似于过程（Procedure）的创建，但它采用 @UserFunction 注解，并且只返回一个单值。有效的输出类型包括 long、Long、double、Double、boolean、Boolean、String、Node、Relationship、Path、Map<String、Object>或者 List<T>，这里的 T 可以是任意支持的类型。

下面是一个简单的自定义函数例子，该函数将 List 中的字符串用指定的分隔符连接起来。

```
package example;
import org.neo4j.procedure.Name;
import org.neo4j.procedure.Procedure;
import org.neo4j.procedure.UserFunction;
public class Join
{
@UserFunction
@Description("example.join(['s1','s2',...], delimiter) - join the given
strings with the given
```

```
delimiter.")
public String join(
@Name("strings") List<String> strings,
@Name(value = "delimiter", defaultValue = ",") String delimiter) {
  if (strings == null || delimiter == null) {
    return null;
  }
  return String.join(delimiter, strings);
 }
}
```

3.5 模式（Schema）

本小节介绍如何在 Cypher 中使用 Neo4j 的可选模式。

基于标签的概念，Neo4j 2.0 为图引入了可选模式。在索引的规范中的标签为图定义约束。索引和约束是图的模式。Cypher 引入了数据定义语言（Data Definition Language，DDL）来操作模式。

3.5.1 索引

数据库的索引是为了使得检索数据效率更高而引入的冗余信息。它的代价是需要额外的存储空间和使得写入时变得更慢。因此，决定哪些数据需要建立索引，哪些不需要是非常重要的工作。

Cypher 允许所有节点的某个属性上有特定的标签。索引一旦创建，它将自己管理并当图发生变化时自动更新。一旦索引创建并生效之后，Neo4j 将自动开始使用索引。

3.5.1.1 创建索引

使用 CREATE INDEX ON 可以在拥有某个标签的所有节点的某个属性上创建索引。注意，索引是在后台创建，并不能立刻就生效。

查询

```
CREATE INDEX ON :Person(name)
```

本例在拥有 Person 标签的所有节点的 name 属性上创建了索引。



Neo4j 中的标签类似关系数据库中的表名，属性名就相当于表的列名。这里可以看做是在 Person 表的 name 列上建立了索引。

结果

```
+-------------------------------------------+
| No data returned, and nothing was changed. |
+-------------------------------------------+
```

3.5.1.2　删除索引

使用 DROP INDEX 可以删除拥有某个标签的所有节点的某个属性上的索引。

查询

```
DROP INDEX ON :Person(name)
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Indexes removed: 1
```

3.5.1.3　使用索引

通常不需要在查询中指出使用哪个索引，Cypher 自己会决定。例如，如果存在 Person(name)索引，下面的查询将会使用到它。如果希望 Cypher 使用指定的索引，可以使用 USING 来提示，详细参见“3.6.4 USING”小节。

查询

```
MATCH (person:Person { name: 'Andres' })
RETURN person
```

查询计划

```
+-----------------+----------------+------+---------+-----------+---------------+
| Operator        | Estimated Rows | Rows | DB Hits | Variables | Other         |
+-----------------+----------------+------+---------+-----------+---------------+
| +ProduceResults | 1              | 1    | 0       | person    | person        |
| |               +----------------+------+---------+-----------+---------------+
| +NodeIndexSeek  | 1              | 1    | 2       | person    | :Person(name) |
+-----------------+----------------+------+---------+-----------+---------------+
Total database accesses: 2
```

3.5.1.4　在 WHERE 等式中使用索引

在 WHERE 语句中对索引的属性进行相等比较时，索引将自动被使用。

查询

```
MATCH (person:Person)
WHERE person.name = 'Andres'
RETURN person
```

查询计划

```
+-----------------+----------------+------+---------+-----------+---------------+
| Operator        | Estimated Rows | Rows | DB Hits | Variables | Other         |
+-----------------+----------------+------+---------+-----------+---------------+
| +ProduceResults |    1           | 1    |  0      | person    | person        |
| |               +----------------+------+---------+-----------+---------------+
| +NodeIndexSeek  |    1           | 1    |  2      | person    | :Person(name) |
+-----------------+----------------+------+---------+-----------+---------------+
Total database accesses: 2
```

3.5.1.5 使用 WHERE 不等式中使用索引

在 WHERE 语句中对索引的属性进行不等（范围）比较时，索引将自动被使用。

查询

```
MATCH (person:Person)
WHERE person.name > 'B'
RETURN person
```

查询计划

```
+----------------------+-----------+------+------+-----------+-----------------+
| Operator             | Estimated | Rows | DB   | Variables | Other           |
|                      | Rows      |      | Hits |           |                 |
+----------------------+-----------+------+------+-----------+-----------------+
| +ProduceResults      |        10 |    1 |    0 | person    | person          |
| |                    +-----------+------+------+-----------+-----------------+
| +NodeIndexSeekByRange|        10 |    1 |    2 | person    | :Person(name) > |
|                      |           |      |      |           | { AUTOSTRING0}  |
+----------------------+-----------+------+------+-----------+-----------------+
Total database accesses: 2
```

3.5.1.6 在 IN 中使用索引

下面查询中针对 person.name 的 IN 断言将使用 Person(name)索引。

查询

```
MATCH (person:Person)
WHERE person.name IN ['Andres', 'Mark']
RETURN person
```

查询计划

```
+-----------------+----------------+------+---------+-----------+---------------+
| Operator        | Estimated Rows | Rows | DB Hits | Variables | Other         |
+-----------------+----------------+------+---------+-----------+---------------+
| +ProduceResults |             24 |    2 |       0 | person    | person        |
| |               +----------------+------+---------+-----------+---------------+
| +NodeIndexSeek  |             24 |    2 |       4 | person    | :Person(name) |
+-----------------+----------------+------+---------+-----------+---------------+
Total database accesses: 4
```

3.5.1.7 在 STARTS WITH 中使用索引

下面的查询语句在针对 person.name 的 STARTS WITH 断言将使用 Person(name)索引。

查询

```
MATCH (person:Person)
WHERE person.name STARTS WITH 'And'
RETURN person
```

查询计划

```
+----------------------+-----------+------+------+-----------+--------------------+
| Operator             | Estimated | Rows | DB   | Variables | Other              |
```

```
|                        | Rows      |      | Hits |           |                      |
+------------------------+-----------+------+------+-----------+----------------------+
| +ProduceResults        |      26   |   1  |   0  | person    | person               |
| |                      +-----------+------+------+-----------+----------------------+
| +NodeIndexSeekByRange  |      26   |   1  |   2  | person    | :Person(name STARTS  |
|                        |           |      |      |           | WITH {AUTOSTRING0}) |
+------------------------+-----------+------+------+-----------+----------------------+
Total database accesses: 2
```

3.5.1.8　在检查属性存在性时使用索引

下面查询中的 has(p.name)断言将使用 Person(name)索引。

查询

```
MATCH (p:Person)
WHERE exists(p.name)
RETURN p
```

查询计划

```
+-----------------+----------------+------+---------+-----------+---------------+
| Operator        | Estimated Rows | Rows | DB Hits | Variables | Other         |
+-----------------+----------------+------+---------+-----------+---------------+
| +ProduceResults |              2 |    2 |       0 |         p |             p |
| |               +----------------+------+---------+-----------+---------------+
| +NodeIndexScan  |              2 |    2 |       3 |         p | :Person(name) |
+-----------------+----------------+------+---------+-----------+---------------+
Total database accesses: 3
```

3.5.2　约束

Neo4j 通过使用约束来保证数据完整性。约束可应用于节点或者关系，可以创建节点属性的唯一性约束，也可以创建节点和关系的属性存在性约束。

可以使用属性的唯一性约束确保拥有特定标签的所有节点的属性的值是唯一的，唯一性约束并不意味着所有的节点的属性都必须有一个唯一的值。这个规则不适用于没有该属性的节点。

可以使用属性的存在性约束确保拥有特定标签的所有节点或者拥有特定类型的所有关系的属性是存在的，所有的试图创建新的没有该属性的节点或关系，以及试图删除强制属性的查询都将失败。



只有 Neo4j 企业版才具有属性存在性约束这个高级功能。

可以对某个给定的标签添加多个约束，也可以将唯一性约束和存在性约束同时添加到同一个属性上。

记住，添加约束是一个需要花一段时间的原子操作。在所有已存在的数据被扫描完之前，Neo4j 不会让这个约束生效。



在属性上添加唯一性约束的时候，同时也会自动为该属性添加一个索引。因此，不能单独地添加这样一个索引。Cypher 将像其他索引一样正常使用这种索引，如果想删除唯一性约束但保留该属性上的索引，只有另外创建一个索引可以解决这个问题。

3.5.2.1 节点属性的唯一性约束

1. 创建唯一性约束

使用 IS UNIQUE 语法创建约束，它能确保数据库中拥有特定标签和属性值的节点是唯一的。

查询

```
CREATE CONSTRAINT ON (book:Book) ASSERT book.isbn IS UNIQUE
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Unique constraints added: 1
```

2. 删除唯一性约束

使用 DROP CONSTRAINT 可以删除数据库中的一个约束。

查询

```
DROP CONSTRAINT ON (book:Book) ASSERT book.isbn IS UNIQUE
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Unique constraints removed: 1
```

3. 创建遵从属性唯一性约束的节点

创建一个数据库中还不存在的 isbn 的 Book 节点。

查询

```
CREATE (book:Book { isbn: '1449356265', title: 'Graph Databases' })
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 1
Properties set: 2
Labels added: 1
```

4. 创建违背属性唯一性约束的节点

创建一个数据库中已经存在的 isbn 的节点。

查询

```
CREATE (book:Book { isbn: '1449356265', title: 'Graph Databases' })
```

这种情况下，节点将创建失败。

错误消息

```
Node 0 already exists with label Book and property "isbn"=[1449356265]
```

5. 因为冲突的节点而创建属性唯一性约束失败

当数据库中已经有两个 Book 节点拥有相同的 isbn 号时，在 Book 节点的 isbn 属性上创建属性唯一性约束。

查询

```
CREATE CONSTRAINT ON (book:Book) ASSERT book.isbn IS UNIQUE
```

这种情况下约束将创建失败，因为它与已有的数据冲突。可以选择创建索引或者移除冲突的节点，然后再重新创建约束。

错误消息

```
Unable to create CONSTRAINT ON ( book:Book ) ASSERT book.isbn IS UNIQUE:
Multiple nodes with label `Book` have property `isbn` = '1449356265':
node(0)
node(1)
```

3.5.2.2　节点属性存在性约束

1. 创建节点属性存在性约束

使用 ASSERT exists(variable.propertyName)创建约束，可确保有指定标签的所有节点都有一个特定的属性。

查询

```
CREATE CONSTRAINT ON (book:Book) ASSERT book.isbn IS UNIQUE
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Property existence constraints added: 1
```

2. 删除节点属性存在性约束

使用 DROP CONSTRAINT 可以从数据库中移除一个约束。

查询

```
DROP CONSTRAINT ON (book:Book) ASSERT exists(book.isbn)
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Property existence constraints removed: 1
```

3. 创建遵从属性存在性约束的节点

创建一个存在 isbn 属性的 Book 节点。

查询

```
CREATE (book:Book { isbn: '1449356265', title: 'Graph Databases' })
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 1
Properties set: 2
Labels added: 1
```

4. 创建违背属性存在性约束的节点

在:Book(isbn)有存在性约束的情况下，试图创建一个没有 isbn 属性的 Book 节点。

查询

```
CREATE (book:Book { title: 'Graph Databases' })
```

这种情况下节点将创建失败。

错误消息

```
Node 1 with label "Book" must have the property "isbn" due to a constraint
```

5. 删除有存在性约束的节点属性

在:Book(isbn)有存在性约束的情况下，试图从一个已存在的 Book 节点移除 isbn 属性。

查询

```
MATCH (book:Book { title: 'Graph Databases' })
REMOVE book.isbn
```

这种情况下，移除属性将失败。

错误消息

```
Node 0 with label "Book" must have the property "isbn" due to a constraint
```

6. 因已存在的节点而创建节点属性存在性约束失败

当数据库中存在 Book 节点没有 isbn 属性时，试图在 Book 标签节点的 isbn 属性上创建属性存在性约束。

查询

```
CREATE CONSTRAINT ON (book:Book) ASSERT exists(book.isbn)
```

这种情况因为与已存在的数据冲突，因此约束创建失败。可以选择移除冲突的节点，然后再重新创建约束。

错误消息

```
Unable to create CONSTRAINT ON ( book:Book ) ASSERT exists(book.isbn):
Node(0) with label `Book` has no value for property `isbn`
```

3.5.2.3　关系属性存在性约束

1. 创建关系属性存在性约束

使用 ASSERT exists(variable.propertyName)创建约束，可确保特定类型的所有关系都有一个特定的属性。

查询

```
CREATE CONSTRAINT ON ()-[like:LIKED]-() ASSERT exists(like.day)
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Property existence constraints added: 1
```

2. 删除关系属性存在性约束

使用 DROP CONSTRAINT 从数据库中移除一个约束。

查询

```
DROP CONSTRAINT ON ()-[like:LIKED]-() ASSERT exists(like.day)
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Property existence constraints removed: 1
```

3. 创建遵从属性存在性约束的关系

创建一个存在 day 属性的 LIKED 关系。

查询

```
CREATE (user:User)-[like:LIKED { day: 'yesterday' }]->(book:Book)
```

结果

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 2
Relationships created: 1
Properties set: 1
Labels added: 2
```

4. 创建违背属性存在性约束的关系

在有:LIKED(day)存在性约束的情况下，试图创建一个没有 day 属性的 LIKED 关系。

查询

```
CREATE (user:User)-[like:LIKED]->(book:Book)
```

这种情况下，关系将创建失败。

错误消息

```
Relationship 1 with type "LIKED" must have the property "day" due to a
constraint
```

5. 移除具有存在性约束的关系属性

有:LIKED(day)存在性约束的情况下，试图从一个已有 LIKED 关系中移除 day 属性。

查询

```
MATCH (user:User)-[like:LIKED]->(book:Book)
REMOVE like.day
```

错误消息

```
Relationship 0 with type "LIKED" must have the property "day" due to a
constraint
```

6. 因已存在的关系而创建关系属性存在性约束失败

当数据库中存在 LIKED 关系没有 day 属性时，试图在 LIKED 关系的 day 属性上创建属性存在性约束。

查询

```
CREATE CONSTRAINT ON ()-[like:LIKED]-() ASSERT exists(like.day)
```

这种情况因为与已存在的数据冲突，因此约束创建失败。可以选择移除冲突的关系，然后再重新创建约束。

错误消息

```
Unable to create CONSTRAINT ON ()-[ liked:LIKED ]-() ASSERT exists(liked.day):
Relationship(0) with type `LIKED` has no value for property `day`
```

3.5.3　统计

当执行一个 Cypher 查询时，它将先编译为一个执行计划（Execution Plan），该计划可以运行并响应查询。为了给查询提供一个高效的计划，Neo4j 需要数据库的信息，如模式（Schema）有什么索引和约束存在？Neo4j 也使用统计信息来保持数据库优化执行计划。有了这些信息，Neo4j 就能决定采用哪种模式将获得最好的执行计划。

需要的统计信息如下：

- 拥有特定标签的节点的数量。
- 每个索引的可选择性。
- 按类型分的关系的数量。
- 以拥有指定标签的节点开始或者结束的关系，按类型分各自的数量。

Neo4j 以两种方式来保持这些统计信息的更新。以标签数量为例，每当设置或者删除一个节点的标签的时候，这些数量都会被更新。Neo4j 需要扫描所有索引以获得可选择的数量。

3.5.3.1　配置选项

当产生执行计划的统计信息发生变化时，缓存的执行计划将被重新生成。下面的配置项可以控制执行计划的更新。

（1）dbms.index_sampling.background_enabled

控制当需要更新时索引是否会自动重新采样。Cypher 查询计划器依赖于准确的统计信息来创建执行计划，因此当数据库更新时保持同步是很重要的。



如果后台采样被关闭了，请确保当数据有更新操作时触发手动采样。

（2）dbms.index_sampling.update_percentage

控制多大比例的索引被更新后才触发新的采样？

（3）cypher.statistics_divergence_threshold

控制一个执行计划被认为过时并必须重新生成前，允许多少统计信息发生变化。任何统计信息的相对变化超过临界值，原有执行计划将被丢弃并创建一个新的计划。临界值 0.0 意味着有变化就更新，1.0 意味着永远都不更新。

3.5.3.2　手动索引采样

索引重采样可使用内嵌的 db.resampleIndex()和 db.resampleOutdatedIndexes()两个内嵌过程来触发。

下面是使用 cypher-shell 触发重采样的例子：

- cypher-shell 'CALL db.resampleIndex(":Person(name)");'
- cypher-shell 'CALL db.resampleOutdatedIndexes();'

3.6 查询调优

本节描述 Cypher 查询语言的调优。

Neo4j 会尽可能快地执行查询，然而利用这个领域的知识重新组织查询语句以获取最好的性能还是很有用处的。手动查询性能优化的总目标是确保只从图中检索必要的数据。不必要的数据应该尽可能早地被过滤掉，以减少查询执行后期处理的数据量。避免返回整个节点和关系，而只选取节点和关系中需要的数据。同时，也应该设置可变长度模式的一个上限值，以避免包含大量不需要的数据部分。

Cypher 执行引擎会将每个 Cypher 查询都转为一个执行计划。为了减少使用的资源，如果可能，应尽可能地使用参数代替字面值。这会使得 Cypher 可以重用查询，而不必解析并构建一个新的执行计划。本小节提到的关于执行计划运算符的更多信息，可参见“3.7 执行计划”一节。

3.6.1 查询如何执行

每个查询都被查询计划器转化为一个执行计划。在执行查询时，执行计划将告知 Neo4j 执行什么样的操作。Neo4j 包括两种不同的执行计划策略：

1. 规则

查询计划器有用于产生查询计划的规则。它会考虑所有可用的索引，但不使用统计信息去指导查询编译。

2. 成本

计划器使用统计服务为所有可选的查询赋予一个成本，然后选耗费最少的那个。这在大多数情况下会得到更优的查询计划，但这个功能还在开发中。

默认情况下，Neo4j 3.1 对所有查询使用成本计划器。可以通过使用 cypher.planner 配置来强制使用指定的计划器。可以通过查看执行计划看到使用的哪个计划器。



当 Cypher 构建执行计划时，它会检查模式看是否能找到可用的索引。这些索引决策一直有效直到数据模式发生变化。所以，增加和删除索引都会导致缓存的执行计划被刷新。

3.6.2　查询性能分析

查看执行计划对查询进行分析时有两个选项：

1. EXPLAIN

如果只想查看查询计划，而不想运行该语句，可以在查询语句中加入 EXPLAIN。此时，该语句将返回空结果，对数据库不会做出任何改变。

2. PROFILE

如果想运行查询语句并查看哪个运算符占了大部分的工作，可以使用 PROFILE。此时，该语句将被运行，并跟踪传递了多少行数据给每个运算符，以及每个运算符与存储层交互了多少以获取必要的数据。注意，加入 PROFILE 的查询语句将占用更多的资源，所以除非真正在做性能分析，否则不要使用 PROFILE。



在查询中显式地指出返回的关系的类型和节点的标签，将有助于 Neo4j 使用最合适的统计信息，这样会产生更好的执行计划。这意味着，当知道要查询关系的类型时，应该在查询中添加进去。同样对于节点，在关系的开始和结束节点中指明标签将有助于 Neo4j 找到执行语句的最佳方式。

3.6.3　查询调优举例

这里通过一个简单的例子来熟悉查询调优。下面的例子将使用电影数据集，先导入数据：

```
LOAD CSV WITH HEADERS FROM 'http://neo4j.com/docs/developer-
manual/3.1/csv/query-tuning/movies.csv' AS line
MERGE (m:Movie { title: line.title })
ON CREATE SET m.released = toInt(line.released), m.tagline = line.tagline
LOAD CSV WITH HEADERS FROM 'http://neo4j.com/docs/developer-
manual/3.1/csv/query-tuning/actors.csv' AS line
MATCH (m:Movie { title: line.title })
MERGE (p:Person { name: line.name })
ON CREATE SET p.born = toInt(line.born)
MERGE (p)-[:ACTED_IN { roles:split(line.roles, ';')}]->(m)
LOAD CSV WITH HEADERS FROM 'http://neo4j.com/docs/developer-
manual/3.1/csv/query-tuning/directors.csv' AS line
MATCH (m:Movie { title: line.title })
MERGE (p:Person { name: line.name })
ON CREATE SET p.born = toInt(line.born)
MERGE (p)-[:DIRECTED]->(m)
```

写一个找到'Tom Hanks'的查询语句。比较初级的做法是按如下方式写：

```
MATCH (p { name: 'Tom Hanks' })
RETURN p
```

这个查询将会找到'Tom Hanks'节点，但是随着数据库中节点数的增加，该查询将越来越慢。可以通过使用 PROFILE 来找到原因。

可以在“3.6.2 查询性能分析”小节获得更多关于查询调优选项的信息。这里只是在查询前面加入一个 PROFILE 的前缀。

```
PROFILE
MATCH (p { name: 'Tom Hanks' })
RETURN p
```

结果

```
Compiler CYPHER 3.1
Planner COST
Runtime INTERPRETED
+-----------------+-----------+------+------+-----------+-------------------------+
| Operator        | Estimated | Rows | DB   | Variables | Other                   |
|                 | Rows      |      | Hits |           |                         |
+-----------------+-----------+------+------+-----------+-------------------------+
| +ProduceResults |        16 |    1 |    0 |         p | p                       |
| |               +-----------+------+------+-----------+-------------------------+
| +Filter         |        16 |    1 |  163 |         p | p.name == { AUTOSTRING0}|
| |               +-----------+------+------+-----------+-------------------------+
| +AllNodesScan   |       163 |  163 |  164 |         p |                         |
+-----------------+-----------+------+------+-----------+-------------------------+
Total database accesses: 327
```

首先需要记住的是，查看执行计划应该从底端往上看。在这个过程中，我们注意到从最后一行开始的 Rows 列中的数字远高于给定的 name 属性为'Tom Hanks'的一个节点。在 Operator 列中我们看到 AllNodeScan 被使用到了，这意味着查询计划器扫描了数据库中的所有节点。

向上移动一行看 Filter 运算符，它将检查由 AllNodeScan 传入的每个节点的 name 属性。这看起来是一种非常低效的方式来查找'Tom Hanks'。

解决这个问题的办法是，无论什么时候查询一个节点，都应该指定一个标签来帮助查询计划器缩小搜索空间的范围。对于这个查询，可简单地添加一个 Person 标签。

```
MATCH (p:Person { name: 'Tom Hanks' })
RETURN p
```

这个查询会比第一个查询要快，但随着数据库中人数的增加，它依然变得越来越慢。我们再次使用 ROFILE 来查看原因：

```
PROFILE
MATCH (p:Person { name: 'Tom Hanks' })
RETURN p
```

结果

```
Compiler CYPHER 3.1
Planner COST
Runtime INTERPRETED
+-----------------+-----------+------+------+-----------+-----------------+
| Operator        | Estimated | Rows | DB   | Variables | Other           |
|                 | Rows      |      | Hits |           |                 |
```

```
+------------------+-----------+------+------+----------+--------------------+
| +ProduceResults  |      13   |   1  |   0  |        p | p                  |
| |                +-----------+------+------+----------+--------------------+
| +Filter          |      13   |   1  | 125  |        p |p.name={AUTOSTRING0} |
| |                +-----------+------+------+----------+--------------------+
| +NodeByLabelScan |     125   | 125  | 126  |        p | :Person            |
+------------------+-----------+------+------+----------+--------------------+
Total database accesses: 251
```

这次最后一行 Rows 的值已经降低了，这里没有扫描到之前扫描到的那些节点。NodeByLabelScan 运算符表明首先在数据库中做了一个针对所有 Person 节点的线性扫描。一旦完成后，后续将针对所有节点执行 Filter 运算符，依次比较每个节点的 name 属性。

这在某些情况下看起来还可以接受，但是如果频繁通过 name 属性来查询 Person，针对带有 Person 标签的节点的 name 属性创建索引将获得更好的性能。

```
CREATE INDEX ON :Person(name)
```

现在再次运行该查询将运行得更快。

```
MATCH (p:Person { name: 'Tom Hanks' })
RETURN p
```

通过 PROFILE 来看看为什么：

```
PROFILE
MATCH (p:Person { name: 'Tom Hanks' })
RETURN p
```

结果

```
Compiler CYPHER 3.1
Planner COST
Runtime INTERPRETED
+------------------+----------------+------+---------+-----------+---------------+
| Operator         | Estimated Rows | Rows | DB Hits | Variables | Other         |
+------------------+----------------+------+---------+-----------+---------------+
| +ProduceResults  |              1 |    1 |       0 |         p | p             |
| |                +----------------+------+---------+-----------+---------------+
| +NodeIndexSeek   |              1 |    1 |       2 |         p | :Person(name) |
+------------------+----------------+------+---------+-----------+---------------+
Total database accesses: 2
```

查询计划下降到单一的行并使用了 NodeIndexSeek 运算符，它通过模式索引寻找到对应的节点。

3.6.4　USING

USING 语句用于为一个查询构建执行计划时影响计划器的决定。



强制计划器的操作是一个高级功能，由于它可能会导致查询性能的降低，应该由有经验的开发人员或者数据库管理员谨慎使用。

3.6.4.1 简介

当执行一个查询时，Neo4j 需要决定从查询图中的哪儿开始匹配。这是通过查看 MATCH 语句和 WHERE 中的条件这些信息来找到有用的索引或者其他开始节点。

然而，选定的索引未必总是最好的选择。有时候，查询计划器可能从性能的角度在众多可能的备选索引中选择了错误的一个。在某些情况（尽管很少）下可能最好什么索引也不用。

可以通过 USING 来强制 Neo4j 使用一个特定的开始点，这个被称为计划器提示。这里有三种类型的计划器提示：索引提示、扫描提示和连接（Join）提示。



含有 START 的语句不能使用查询器提示。

后续的例子将基于图 3-28 所示的图例。

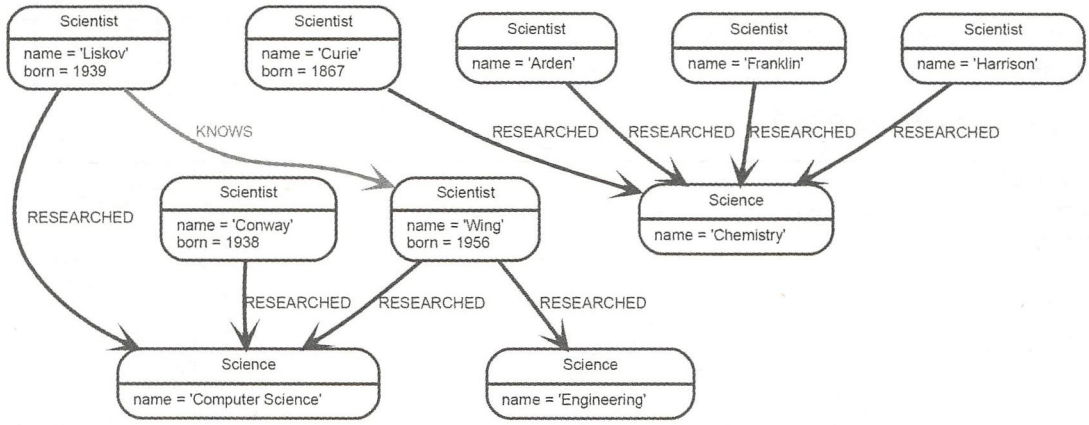


图 3-28　USING 图例

查询

```
MATCH (liskov:Scientist { name:'Liskov' })-[:KNOWS]->(wing:Scientist)-
[:RESEARCHED]->(cs:Science {name:'Computer Science' })<-[:RESEARCHED]-
(conway:Scientist { name: 'Conway' })
RETURN 1 AS column
```

后续的例子将使用此查询。它故意以这种方式构建，这个查询匹配到的子图的统计信息将不准确。因此，可以通过添加查询器提示来优化性能。

查询计划

```
+----------------+-----+----+----+--------------------+-------------------------+
| Operator       |Esti-|Rows|DB  | Variables          | Other                   |
|                |mated|    |Hits|                    |                         |
|                |Rows |    |    |                    |                         |
+----------------+-----+----+----+--------------------+-------------------------+
| +ProduceResults|   0 |  1 |  0 | column             | column                  |
| |              +-----+----+----+--------------------+-------------------------+
```

```
| +Projection    |  0  | 1  |  0  | column -- anon[122],| {column : { AUTOINT3}}  |
| |              |     |    |     |  anon[41],anon[68], |                         |
| |              |     |    |     |  conway, cs,liskov, |                         |
| |              |     |    |     |  wing               |                         |
| |              +-----+----+-----+---------------------+-------------------------+
| +Filter        |  0  | 1  |  3  | anon[122], anon[41],| NOT(anon[122] == anon[68]) |
| |              |     |    |     |  anon[68], conway,  |  AND conway.name ==     |
| |              |     |    |     |  cs, liskov,wing    |  {AUTOSTRING2} AND      |
| |              |     |    |     |                     |  conway:Scientist       |
| |              +-----+----+-----+---------------------+-------------------------+
| +Expand(All)   |  0  | 3  |  4  | anon[122], conway --| (cs)<-[:RESEARCHED]-    |
| |              |     |    |     |  anon[41],anon[68], |  (conway)               |
| |              |     |    |     |   cs, liskov,wing   |                         |
| |              +-----+----+-----+---------------------+-------------------------+
| +Filter        |  0  | 1  |  4  | anon[41], anon[68], | cs:Science AND cs.name  |
| |              |     |    |     |  cs,liskov, wing    |   == { AUTOSTRING1}     |
| |              +-----+----+-----+---------------------+-------------------------+
| +Expand(All)   |  0  | 2  |  3  | anon[68], cs --     | (wing)-[:RESEARCHED]->  |
| |              |     |    |     |  anon[41], liskov,  |  (cs)                   |
| |              |     |    |     |  wing               |                         |
| |              +-----+----+-----+---------------------+-------------------------+
| +Filter        |  0  | 1  |  1  | anon[41], liskov,   | wing:Scientist          |
| |              |     |    |     |  wing               |                         |
| |              +-----+----+-----+---------------------+-------------------------+
| +Expand(All)   |  0  | 1  |  2  | anon[41], wing --   |(liskov)-[:KNOWS]->(wing)|
| |              |     |    |     |  liskov             |                         |
| |              +-----+----+-----+---------------------+-------------------------+
| +NodeIndexSeek |  1  | 1  |  2  | liskov              | :Scientist(name)        |
+----------------+-----+----+-----+---------------------+-------------------------+
Total database accesses: 19
```

3.6.4.2　索引提示

索引提示用于告知计划器无论在什么情况下都应使用指定的索引作为开始点。对于某些特定值的查询，索引统计信息不准确，它可能导致计划器选择了非最优的索引。对于这种情况，索引提示就有它的用处。使用在 MATCH 语句之后添加 USING INDEX variable:Label(property)来补充索引提示。

也可以补充多个索引提示，但是多个开始点会在后面的查询计划中潜在地需要额外的连接。

1. 使用索引提示查询

上面的查询没有选择索引来生成计划。这是因为图非常小，对于小数据库标签扫描很快。然而，查询的性能通常以 dbhit 的值来度量。下面可以看到使用索引将获得更好的查询性能。

查询

```
MATCH (liskov:Scientist { name:'Liskov' })-[:KNOWS]->(wing:Scientist)-
[:RESEARCHED]->(cs:Science {name:'Computer Science' })<-[:RESEARCHED]-
(conway:Scientist { name: 'Conway' })
USING INDEX liskov:Scientist(name)
RETURN liskov.born AS column
```

返回'Barbara Liskov'的出生年份。

查询计划

```
+----------------+-----+----+----+--------------------+-------------------------+
| Operator       |Esti-|Rows|DB  | Variables          | Other                   |
|                |mated|    |Hits|                    |                         |
|                |Rows |    |    |                    |                         |
+----------------+-----+----+----+--------------------+-------------------------+
| +ProduceResults|  0  | 1  | 0  | column             | column                  |
| |              +-----+----+----+--------------------+-------------------------+
| +Projection    |  0  | 1  | 1  | column -- anon[122],| {column : liskov.born} |
| |              |     |    |    |  anon[41],anon[68], |                         |
| |              |     |    |    |  conway,cs,liskov,  |                         |
| |              |     |    |    |  wing               |                         |
| |              +-----+----+----+--------------------+-------------------------+
| +Filter        |  0  | 1  | 3  | anon[122], anon[41],| NOT(anon[122]==anon[68])|
| |              |     |    |    |  anon[68], conway,  |  AND conway.name ==     |
| |              |     |    |    |  cs, liskov,wing    |  {AUTOSTRING2} AND      |
| |              |     |    |    |                     |  conway:Scientist       |
| |              +-----+----+----+--------------------+-------------------------+
| +Expand(All)   |  0  | 3  | 4  | anon[122],conway -- | (cs)<-[:RESEARCHED]-    |
| |              |     |    |    |  anon[41],anon[68], |  (conway)               |
| |              |     |    |    |  cs, liskov,wing    |                         |
| |              +-----+----+----+--------------------+-------------------------+
| +Filter        |  0  | 1  | 4  | anon[41], anon[68], | cs:Science AND cs.name==|
| |              |     |    |    |  cs, liskov, wing   |  {AUTOSTRING1}          |
| |              +-----+----+----+--------------------+-------------------------+
| +Expand(All)   |  0  | 2  | 3  | anon[68], cs --     | (wing)-[:RESEARCHED]->  |
| |              |     |    |    |  anon[41], liskov,  |  (cs)                   |
| |              |     |    |    |  wing               |                         |
| |              +-----+----+----+--------------------+-------------------------+
| +Filter        |  0  | 1  | 1  | anon[41],liskov,wing| wing:Scientist          |
| |              +-----+----+----+--------------------+-------------------------+
| +Expand(All)   |  0  | 1  | 2  | anon[41], wing --   | (liskov)-[:KNOWS]->(wing)|
| |              |     |    |    |  liskov             |                         |
| |              +-----+----+----+--------------------+-------------------------+
| +NodeIndexSeek |  1  | 1  | 2  | liskov             | :Scientist(name)        |
+----------------+-----+----+----+--------------------+-------------------------+
Total database accesses: 20
```

2. 使用多个索引提示查询

提供一个索引提示会改变查询的开始点，但查询计划依然是线性的，因为只有一个开始点。如果为计划器提供另外一个索引提示，强制使用两个开始点，匹配的两端各一个。这时将使用 join 运算符来连接两个分支。

查询

```
MATCH (liskov:Scientist { name:'Liskov' })-[:KNOWS]->(wing:Scientist)-
[:RESEARCHED]->(cs:Science {
name:'Computer Science' })<-[:RESEARCHED]-(conway:Scientist { name:
'Conway' })
USING INDEX liskov:Scientist(name)
USING INDEX conway:Scientist(name)
RETURN liskov.born AS column
```

使用更好的计划返回'Barbara Liskov'的出生年份。

查询计划

```
+------------------+-----+-----+------+----------------------+------------------+
| Operator         |Esti-|Rows |DB    | Variables            | Other            |
|                  |mated|     |Hits  |                      |                  |
|                  |Rows |     |      |                      |                  |
+------------------+-----+-----+------+----------------------+------------------+
| +ProduceResults  |  0  |  1  |  0   | column               | column           |
| |                +-----+-----+------+----------------------+------------------+
| +Projection      |  0  |  1  |  1   | column -- anon[122],  |{column:liskov.born}|
| |                |     |     |      |  anon[41], anon[68],  |                  |
| |                |     |     |      |  conway,cs, liskov,   |                  |
| |                |     |     |      |  wing                 |                  |
| |                +-----+-----+------+----------------------+------------------+
| +Filter          |  0  |  1  |  0   | anon[122], anon[41],  | NOT(anon[122] ==  |
| |                |     |     |      |  anon[68], conway,    |  anon[68])       |
| |                |     |     |      |  cs, liskov,wing      |                  |
| |                +-----+-----+------+----------------------+------------------+
| +NodeHashJoin    |  0  |  1  |  0   | anon[41], anon[68],   | cs               |
| |                |     |     |      |  liskov, wing --      |                  |
| |                |     |     |      |  anon[122],conway,cs  |                  |
| |\               +-----+-----+------+----------------------+------------------+
| | +Filter        |  1  |  1  |  1   | anon[122], conway,cs |cs.name={AUTOSTRING1}|
| | |              +-----+-----+------+----------------------+------------------+
| | +Expand(All)   |  1  |  1  |  2   | anon[122],cs --conway|(conway)-[:RESEARCHED]|
| | |              |     |     |      |                      |  ->(cs)          |
| | |              +-----+-----+------+----------------------+------------------+
| | +NodeIndexSeek |  1  |  1  |  2   | conway               | :Scientist(name) |
| |                +-----+-----+------+----------------------+------------------+
| +Filter          |  0  |  1  |  4   |anon[41], anon[68],cs,|cs:Science AND cs.name|
| |                |     |     |      |  liskov, wing        |  == {AUTOSTRING1} |
| |                +-----+-----+------+----------------------+------------------+
| +Expand(All)     |  0  |  2  |  3   | anon[68],cs--anon[41],|(wing)-[:RESEARCHED]|
| |                |     |     |      |  liskov, wing        |  ->(cs)          |
| |                +-----+-----+------+----------------------+------------------+
| +Filter          |  0  |  1  |  1   | anon[41], liskov, wing| wing:Scientist   |
| |                +-----+-----+------+----------------------+------------------+
| +Expand(All)     |  0  |  1  |  2   | anon[41], wing --     |(liskov)-[:KNOWS]->|
| |                |     |     |      |  liskov              |  (wing)          |
| |                +-----+-----+------+----------------------+------------------+
| +NodeIndexSeek   |  1  |  1  |  2   | liskov               | :Scientist(name) |
+------------------+-----+-----+------+----------------------+------------------+
Total database accesses: 18
```

3.6.4.3　扫描提示

如果查询匹配到一个索引的大部分，它可以更快地扫描标签并过滤掉不匹配的节点。通过在 MATCH 语句后面使用 USING SCAN variable:Label 可以做到这一点。它将强制 Cypher 不使用本应使用的索引，而采用标签扫描。

标签扫描提示

使用 USING SCAN 扫描一个标签上的所有节点并过滤结果集，将获得最好的性能。

查询

```
MATCH (s:Scientist)
USING SCAN s:Scientist
WHERE s.born < 1939
RETURN s.born AS column
```

返回 1939 年以前出生的科学家。

查询计划

```
+------------------+------+-----+-------+------------+---------------------------+
| Operator         |Esti- |Rows |DB     | Variables  | Other                     |
|                  |mated |     |Hits   |            |                           |
|                  |Rows  |     |       |            |                           |
+------------------+------+-----+-------+------------+---------------------------+
| +ProduceResults  |  0   | 2   | 0     | column     | column                    |
| |                +------+-----+-------+------------+---------------------------+
| +Projection      |  0   | 2   | 2     | column--s  | {column : s.born}         |
| |                +------+----+----+-----------+-------------------------------+
| +Filter          |  0   | 2   | 7     | s          | AndedPropertyComparable-  |
| |                |      |     |       |            |  Predicates(s,s.born,     |
| |                |      |     |       |            |  s.born < {AUTOINT0})     |
| |                +------+-----+-------+------------+---------------------------+
| +NodeByLabelScan |  7   | 7   | 8     | s          | :Scientist                |
+------------------+------+-----+-------+------------+---------------------------+
Total database accesses: 17
```

3.6.4.4 连接（Join）提示

连接提示是提示中的高级类型。它不是用于找到查询计划的开始点，而是强制在特定的点进行连接。为了查询能够连接来自这些叶节点的两个分支，这意味着在计划中不止一个开始点（叶节点）。基于这一点，连接和后续的连接提示将强制计划器查看额外的开始点。在那些没有更多好的开始点的情况下，可能会选取一些很差的开始点。这将对查询性能产生负面的效果。在其他情况下，这些提示会强制计划器选取一些看似不好的开始点，然后它会被证明是好的。

1. 提示在单个节点上的连接

本例中使用了多个索引提示，可以看到计划器在 cs 节点做了一个连接。这意味着遍历了 wing 和 cs 之间的外向关系，它能更好地统计，因为模式()-[:RESEARCHED]→(:Science)比模式(:Scientist)-[:RESEARCHED]→()更常用。然而，在实际的图中，cs 节点仅有两个这样的关系，因此从它展开将受益于从 wing 节点展开。可以强制在 wing 上连接而不是使用连接提示。

查询

```
MATCH (liskov:Scientist { name:'Liskov' })-[:KNOWS]->(wing:Scientist)-
[:RESEARCHED]->(cs:Science {
name:'Computer Science' })<-[:RESEARCHED]-(conway:Scientist { name:
'Conway' })
USING INDEX liskov:Scientist(name)
USING INDEX conway:Scientist(name)
USING JOIN ON wing
RETURN wing.born AS column
```

使用更好的查询计划返回'Jeanette Wing'的出生日期。

结果

```
+-----------------+--------+--------+-------+--------------------+----------------+
| Operator        |Esti-   |Rows    |DB     | Variables          | Other          |
|                 |mated   |        |Hits   |                    |                |
|                 |Rows    |        |       |                    |                |
+-----------------+--------+--------+-------+--------------------+----------------+
| +ProduceResults |  0     | 1      | 0     | column             | column         |
| |               +--------+--------+-------+--------------------+----------------+
| +Projection     |  0     | 1      | 1     | column--anon[122],| {column:wing.born}|
| |               |        |        |       | anon[41],anon[68], |                |
| |               |        |        |       | conway,cs, liskov, |                |
| |               |        |        |       |  wing              |                |
| |               +--------+--------+-------+--------------------+----------------+
| +NodeHashJoin   |  0     | 1      | 0     |anon[41],liskov--   | wing           |
| |               |        |        |       |anon[122],anon[68], |                |
| |               |        |        |       |conway,cs, wing     |                |
| |\              +--------+--------+-------+--------------------+----------------+
| | +Filter       |  1     | 2      | 0     |anon[122],anon[68], | NOT(anon[122]==|
| | |             |        |        |       |conway, cs, wing    |  anon[68])     |
| | |             +--------+--------+-------+--------------------+----------------+
| | +Expand(All)  |  1     | 3      | 4     |anon[68],wing--     |(cs)<-[:RESEARCHED]|
| | |             |        |        |       |anon[122],conway,cs | -(wing)        |
| | |             +--------+--------+-------+--------------------+----------------+
| | +Filter       |  0     | 1      | 2  |anon[122],conway,cs|cs.name={AUTOSTRING1}|
| | |             |        |        |       |                    |AND cs:Science  |
| | |             +--------+--------+-------+--------------------+----------------+
| | +Expand(All)  |  1     | 1      | 2     | anon[122],cs --  |(conway)-[:RESEARCHED]|
| | |             |        |        |       |  conway            |   ->(cs)       |
| | |             +--------+--------+-------+--------------------+----------------+
| | +NodeIndexSeek|  1     | 1      | 2     | conway             |:Scientist(name)|
| |               +--------+--------+-------+--------------------+----------------+
| +Filter         |  0     | 1      | 1     | anon[41], liskov,wing|wing:Scientist|
| |               +--------+--------+-------+--------------------+----------------+
| +Expand(All)    |  0     | 1      | 2     | anon[41], wing -- | (liskov)-[:KNOWS]->|
| |               |        |        |       |  liskov            |  (wing)        |
| |               +--------+--------+-------+--------------------+----------------+
| +NodeIndexSeek  |  1     | 1      | 2     | liskov             | :Scientist(name)|
+-----------------+--------+--------+-------+--------------------+----------------+
Total database accesses: 16
```

2. 提示在多个节点上的连接

查询计划器可在多个指定的节点上产生一个连接。这要求查询从同一个节点的多个方向展开。

查询

```
MATCH (liskov:Scientist { name:'Liskov' })-[:KNOWS]->(wing:Scientist
{ name:'Wing' })-[:RESEARCHED]->(cs:Science { name:'Computer Science' })<-
[:RESEARCHED]-(liskov)
USING INDEX liskov:Scientist(name)
USING JOIN ON liskov, cs
RETURN wing.born AS column
```

返回'Jeanette Wing'的出生日期。

查询计划

```
+------------------+------+-----+----+----------------------+-------------------+
| Operator         |Esti- |Rows |DB  | Variables            | Other             |
|                  |mated |     |Hits|                      |                   |
|                  |Rows  |     |    |                      |                   |
+------------------+------+-----+----+----------------------+-------------------+
| +ProduceResults  |  0   |  1  |  0 | column               | column            |
| |                +------+-----+----+----------------------+-------------------+
| +Projection      |  0   |  1  |  1 | column -- anon[136], | {column:wing.born}|
| |                |      |     |    |  anon[41], anon[82], |                   |
| |                |      |     |    |  cs,liskov, wing     |                   |
| |                +------+-----+----+----------------------+-------------------+
| +Filter          |  0   |  1  |  0 | anon[136], anon[41], | NOT(anon[136] ==  |
| |                |      |     |    |  anon[82], cs,       |  anon[82])        |
| |                |      |     |    |  liskov, wing        |                   |
| |                +------+-----+----+----------------------+-------------------+
| +NodeHashJoin    |  0   |  1  |  0 | anon[41], anon[82],  | liskov, cs        |
| |                |      |     |    |  wing -- anon[136],  |                   |
| |                |      |     |    |  cs, liskov          |                   |
| |\               +------+-----+----+----------------------+-------------------+
| | +Filter        |  1   |  1  |  2 | anon[136], cs,liskov| cs.name=={AUTOSTRING2}|
| | |              |      |     |    |                      |  AND cs:Science   |
| | |              +------+-----+----+----------------------+-------------------+
| | +Expand(All)   |  1   |  1  |  2 | anon[136],cs--liskov|(liskov)-[:RESEARCHED]|
| | |              |      |     |    |                      |  ->(cs)           |
| | |              +------+-----+----+----------------------+-------------------+
| | +NodeIndexSeek |  1   |  1  |  2 | liskov               | :Scientist(name)  |
| |                +------+-----+----+----------------------+-------------------+
| +Filter          |  0   |  1  |  4 |anon[41],anon[82],cs,|cs:Science AND cs.name|
| |                |      |     |    |  liskov, wing        |   == {AUTOSTRING2}|
| |                +------+-----+----+----------------------+-------------------+
| +Expand(All)     |  0   |  2  |  3 | anon[82],cs--anon[41],|(wing)-[:RESEARCHED] |
| |                |      |     |    |  liskov, wing        |  ->(cs)           |
| |                +------+-----+----+----------------------+-------------------+
| +Filter          |  0   |  1  |  2 | anon[41],liskov,wing | wing:Scientist AND|
| |                |      |     |    |                      |  wing.name        |
| |                |      |     |    |                      |  == {AUTOSTRING1} |
| |                +------+-----+----+----------------------+-------------------+
| +Expand(All)     |  0   |  1  |  2 | anon[41], wing --    |(liskov)-[:KNOWS]->|
| |                |      |     |    |  liskov              |  (wing)           |
| |                +------+-----+----+----------------------+-------------------+
| +NodeIndexSeek   |  1   |  1  |  2 | liskov               | :Scientist(name)  |
+------------------+------+-----+----+----------------------+-------------------+
Total database accesses: 20
```

3.7 执行计划

本小节主要描述执行计划（Execution Plan）中的运算符。

Neo4j 将执行一个查询的任务分解为一些被称为运算符的小块，每个运算符负责整个查询中的一小部分。这些以模式（Pattern）形式连接在一起的运算符被称为一个执行计划。

每个运算符用如下统计信息来注解：

- Rows: Row 运算符产生的行数，只有带有 profile 的查询才有。
- EstimatedRows: 如果 Neo4j 使用基于成本的编译器，可以看到由运算符所产生的预估的行数。编译器使用这个估值来选择合适的执行计划。

- DbHits: 每个运算符都会向 Neo4j 存储引擎请求像检索或者更新数据这样的工作。一次数据库命中（DataDase Hit）是存储引擎工作的一个抽象单位。

如何查看查询的执行计划，请参见“3.6 查询调优”一节。请记住，查询所实际运行数据库的统计信息将决定使用的执行计划。这不保证特定的查询一直使用同一个执行计划。

3.7.1　开始点运算符

这些运算符用于找到图的开始点。

3.7.1.1　全节点扫描

从节点库中读取所有节点。实参中的变量将包含所有这些节点。如果查询中使用这个运算符，在任何大点的数据库中就会遭遇性能问题。

查询

```
MATCH (n)
RETURN n
```

查询计划

```
+-----------------+----------------+------+---------+-----------+-------+
| Operator        | Estimated Rows | Rows | DB Hits | Variables | Other |
+-----------------+----------------+------+---------+-----------+-------+
| +ProduceResults |             35 |   35 |       0 |         n |     n |
| |               +----------------+------+---------+-----------+-------+
| +AllNodesScan   |             35 |   35 |      36 |         n |       |
+-----------------+----------------+------+---------+-----------+-------+
Total database accesses: 36
```

3.7.1.2　通过 id 搜索有向关系

从关系库中通过 id 来读取一个或多个关系将返回关系和两端的节点。

查询

```
MATCH (n1)-[r]->()
WHERE id(r)= 0
RETURN r, n1
```

查询计划

```
+------------------+------+-----+------+-------------+------------------------+
| Operator         |Esti- |Rows |DB    | Variables   | Other                  |
|                  |mated |     |Hits  |             |                        |
|                  |Rows  |     |      |             |                        |
+------------------+------+-----+------+-------------+------------------------+
| +ProduceResults  |  1   |  1  |  0   | n1, r       | r, n1                  |
| |                +------+-----+------+-------------+------------------------+
| +DirectedRelation|   1  |  1  |  1   |anon[17],n1,r|EntityByIdRhs(SingleSeekArg|
| shipByIdSeekPipe |      |     |      |             |  ({AUTOINT0}))         |
+------------------+------+-----+------+-------------+------------------------+
Total database accesses: 1
```

3.7.1.3 通过 id 寻找节点

从节点库中通过 id 读取一个或多个节点。

查询

```
MATCH (n)
WHERE id(n)= 0
RETURN n
```

查询计划

```
+----------------+----------------+------+---------+-----------+-------+
| Operator       | Estimated Rows | Rows | DB Hits | Variables | Other |
+----------------+----------------+------+---------+-----------+-------+
| +ProduceResults |              1 |    1 |       0 |         n |     n |
| |              +----------------+------+---------+-----------+-------+
| +NodeByIdSeek  |              1 |    1 |       1 |         n |       |
+----------------+----------------+------+---------+-----------+-------+
Total database accesses: 1
```

3.7.1.4 通过标签扫描检索节点

使用标签索引，从节点的标签索引中获取拥有指定标签的所有节点。

查询

```
MATCH (person:Person)
RETURN person
```

查询计划

```
+----------------+----------------+------+---------+-----------+---------+
| Operator       | Estimated Rows | Rows | DB Hits | Variables | Other   |
+----------------+----------------+------+---------+-----------+---------+
| +ProduceResults |             14 |   14 |       0 |    person | person  |
| |              +----------------+------+---------+-----------+---------+
| +NodeByLabelScan |             14 |   14 |      15 |    person | :Person |
+----------------+----------------+------+---------+-----------+---------+
Total database accesses: 15
```

3.7.1.5 通过索引检索节点

使用索引搜索节点，节点变量和使用的索引在运算符的实参中。如果索引是一个唯一性索引，运算符将由一个被称为 NodeUniqueIndexSeek 的替代。

查询

```
MATCH (location:Location { name: 'Malmo' })
RETURN location
```

结果

```
+----------------+----------------+------+---------+-----------+----------------+
| Operator       | Estimated Rows | Rows | DB Hits | Variables | Other          |
+----------------+----------------+------+---------+-----------+----------------+
| +ProduceResults |              1 |    1 |       0 |  location | location       |
| |              +----------------+------+---------+-----------+----------------+
| +NodeIndexSeek |              1 |    1 |       2 |  location | :Location(name) |
+----------------+----------------+------+---------+-----------+----------------+
Total database accesses: 2
```

3.7.1.6　通过索引范围（Range）寻找节点

使用索引检索节点，节点的属性值满足给定的字符串前缀。这个运算符可用于 STARTS WITH 和比较符号，如<、>和>=。

查询

```
MATCH (l:Location)
WHERE l.name STARTS WITH 'Lon'
RETURN l
```

查询计划

```
+----------------+-----+----+----+-------+-----------------------------------------+
| Operator       |Esti-|Rows|DB  | Vari- | Other                                   |
|                |mated|    |Hits| ables |                                         |
|                |Rows |    |    |       |                                         |
+----------------+-----+----+----+-------+-----------------------------------------+
| +ProduceResults|  26 |  1 |  0 |     l | l                                       |
| |              +-----+----+----+-------+-----------------------------------------+
| +NodeIndexSeek |  26 |  1 |  2 |     l | :Location(name STARTS WITH {AUTOSTRING0}) |
|  ByRange       |     |    |    |       |                                         |
+----------------+-----+----+----+-------+-----------------------------------------+
Total database accesses: 2
```

3.7.1.7　通过索引包含（Contains）扫描检索节点

一个节点的包含扫描将遍历存储在索引中的所有值，搜索实体中是否包含指定的字符串。这个比索引检索要慢，因为需要检查所有的实体，但也比直接通过标签扫描然后过滤属性库要更快一些。

查询

```
MATCH (l:Location)
WHERE l.name CONTAINS 'al'
RETURN l
```

查询计划

```
+------------------------+-----------+------+------+-----------+------------------+
| Operator               | Estimated | Rows | DB   | Variables |Other             |
|                        | Rows      |      | Hits |           |                  |
+------------------------+-----------+------+------+-----------+------------------+
| +ProduceResults        |        10 |    2 |    0 |         l | l                |
| |                      +-----------+------+------+-----------+------------------+
| +NodeIndexContainsScan |        10 |    2 |    3 |         l | :Location(name); |
|                        |           |      |      |           |  { AUTOSTRING0}  |
+------------------------+-----------+------+------+-----------+------------------+
Total database accesses: 3
```

3.7.1.8　通过索引扫描检索节点

索引扫描将遍历存储在索引中的所有值，它可以找到拥有特定标签和特定属性的所有节点（如 exists(n.prop)）。

查询

```
M MATCH (l:Location)
```

```
WHERE exists(l.name)
RETURN l
```

查询计划

```
+-----------------+----------------+------+---------+-----------+-----------------+
| Operator        | Estimated Rows | Rows | DB Hits | Variables | Other           |
+-----------------+----------------+------+---------+-----------+-----------------+
| +ProduceResults |             10 |   10 |       0 |         l | l               |
| |               +----------------+------+---------+-----------+-----------------+
| +NodeIndexScan  |             10 |   10 |      11 |         l | :Location(name) |
+-----------------+----------------+------+---------+-----------+-----------------+
Total database accesses: 11
```

3.7.1.9 通过 id 寻找无方向关系

从关系库中通过 id 读取一个或多个关系。对于每个关系将返回两行，它们分别为关系的开始节点和结束节点。

查询

```
MATCH (n1)-[r]-()
WHERE id(r)= 1
RETURN r, n1
```

查询计划

```
+-------------------------------+-----------+------+------+---------------+------+
| Operator                      | Estimated | Rows |  DB  | Variables     |Other |
|                               | Rows      |      | Hits |               |      |
+-------------------------------+-----------+------+------+---------------+------+
| +ProduceResults               |         1 |    2 |    0 | n1, r         |r, n1 |
| |                             +-----------+------+------+---------------+------+
| +UndirectedRelationshipByIdSeek|         1 |    2 |    1 | anon[16],n1,r |      |
+-------------------------------+-----------+------+------+---------------+------+
Total database accesses: 1
```

3.7.2 Expand 运算符

这些运算符通过展开图模式来探索图。

3.7.2.1 Expand All

给定一个开始节点，expand-all 将根据关系中的模式沿开始节点或者结束节点展开。它也能处理变长模式的关系。

查询

```
MATCH (p:Person { name: 'me' })-[:FRIENDS_WITH]->(fof)
RETURN fof
```

查询计划

```
+-----------------+-----------+------+------+-------------+---------------------+
| Operator        | Estimated | Rows | DB   |   Variables |     Other           |
|                 | Rows      |      | Hits |             |                     |
+-----------------+-----------+------+------+-------------+---------------------+
| +ProduceResults |         0 |    1 |    0 |         fof |      fof            |
| |               +-----------+------+------+-------------+---------------------+
| +Expand(All)    |         0 |    1 |    2 |  anon[30], | (p)-[:FRIENDS_WITH] |
```

```
| |                |           |      |      |    fof-p   |      ->(fof)          |
| |                +-----------+------+------+------------+-----------------------+
| +NodeIndexSeek   |         1 |    1 |    2 | p          |   :Person(name)       |
+--------------- -+-----------+------+------+------------+-----------------------+
Total database accesses: 4
```

3.7.2.2　Expand Into

当开始和结束节点都已经找到时，expand-into 用于找到两个节点之间连接的所有关系。

查询

```
MATCH (p:Person { name: 'me' })-[:FRIENDS_WITH]->(fof)-->(p)
RETURN fof
```

查询计划

```
+-----------------+-----+----+----+--------------------+----------------------+
| Operator        |Esti-|Rows| DB | Variables          |      Other           |
|                 |mated|    |Hits|                    |                      |
|                 |Rows |    |    |                    |                      |
+-----------------+-----+----+----+--------------------+----------------------+
| +ProduceResults |   0 |  0 |  0 |         fof        |       fof            |
| |               +-----+----+----+--------------------+----------------------+
| +Filter         |   0 |  0 |  0 |    anon[30],       |   NOT(anon[30]       |
| |               |     |    |    |    anon[53], fof, p|     ==anon[53])      |
| |               +-----+----+----+--------------------+----------------------+
| +Expand(Into)   |   0 |  0 |  0 |  anon[30] --       | (p)-[:FRIENDS_WITH]  |
| |               |     |    |    |  anon[53], fof, p  |      ->(fof)         |
| |               +-----+----+----+--------------------+----------------------+
| +Expand(All)    |   0 |  0 |  1 | anon[53], fof -- p |    (p)<--(fof)       |
| |               +-----+----+----+--------------------+----------------------+
| +NodeIndexSeek  |   1 |  1 |  2 |          p         |   :Person(name)      |
+-----------------+-----+----+----+--------------------+----------------------+
Total database accesses: 3
```

3.7.2.3　可选 Expand All

可选 expand 从一个给定节点开始遍历关系，确保在返回结果之前断言得到处理。如果没有找到匹配的关系，则返回 null 并产生一个结束节点变量。

查询

```
MATCH (p:Person)
OPTIONAL MATCH (p)-[works_in:WORKS_IN]->(l)
WHERE works_in.duration > 180
RETURN p, l
```

查询计划

```
+---------------------+-----+----+----+-----------------+-------------------+
| Operator            |Esti-|Rows|DB  |    Variables    |    Other          |
|                     |mated|    |Hits|                 |                   |
|                     |Rows |    |    |                 |                   |
+---------------------+-----+----+----+-----------------+-------------------+
| +ProduceResults     |  14 | 15 |  0 |     l, p        |     p, l          |
| |                   +-----+----+----+-----------------+-------------------+
| +OptionalExpand(All)|  14 | 15 | 44 | l, works_in -- p|  (p)-[works_in    |
| |                   |     |    |    |                 |   :WORKS_IN]->(l) |
| |                   +-----+----+----+-----------------+-------------------+
| +NodeByLabelScan    |  14 | 14 | 15 | p               |    :Person        |
+---------------------+-----+----+----+-----------------+-------------------+
Total database accesses: 59
```


仅供非商业用途或交流学习使用

3.7.3 组合运算符

组合运算符用于将其他运算符拼接在一起。

组合运算符图例如图 3-29 所示。

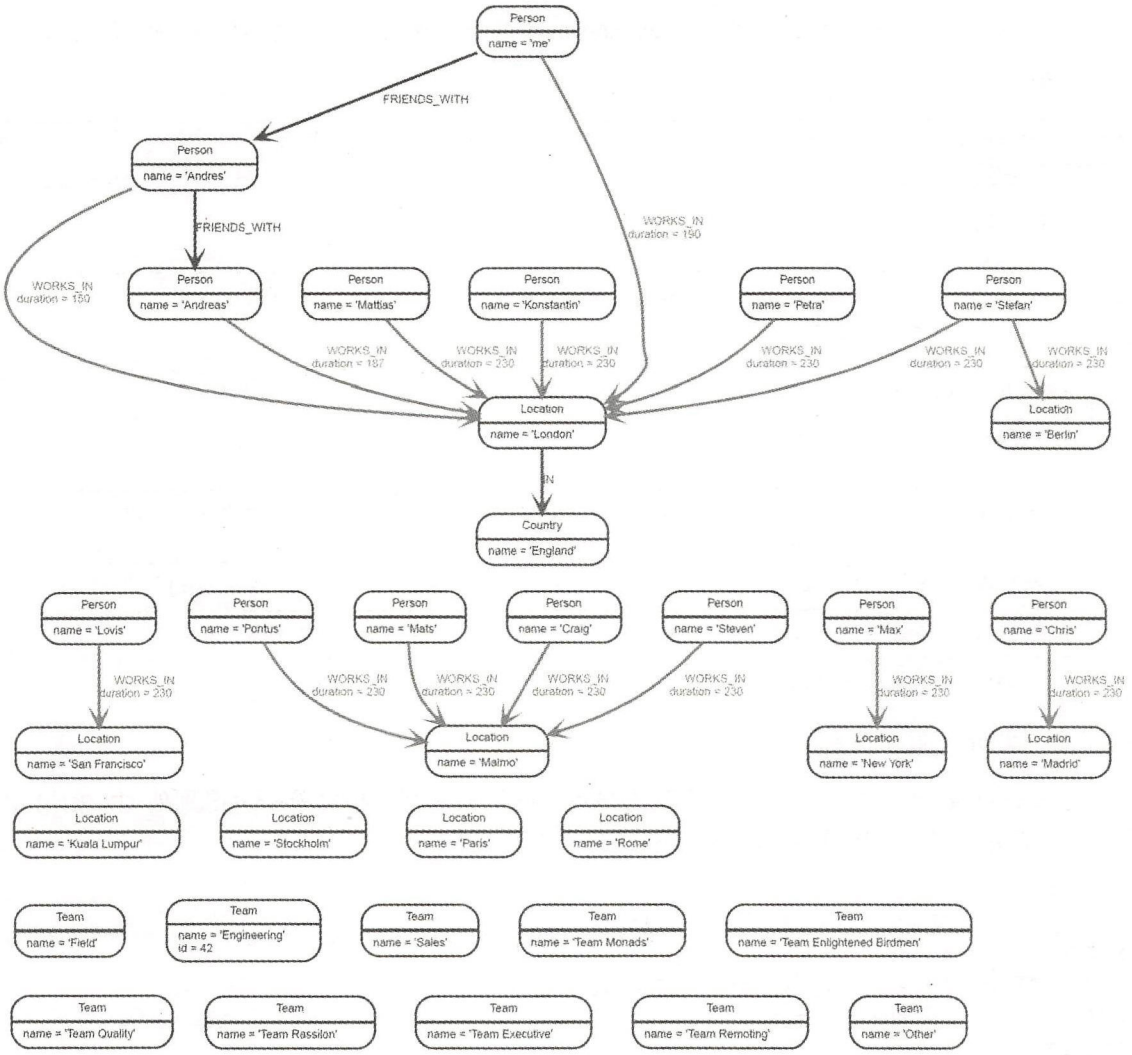


图 3-29 组合运算符图例

3.7.3.1 Apply

Apply 以嵌套循环的方式工作。Apply 运算符左端返回的每一行作为右端运算符的输入，然后 Apply 将产生组合的结果。

查询

```
MATCH (p:Person)-[:FRIENDS_WITH]->(f)
WITH p, count(f) AS fs
```


仅供非商业用途或交流学习使用

```
WHERE fs > 2
OPTIONAL MATCH (p)-[:WORKS_IN]->(city)
RETURN city.name
```

找到多于两个朋友的所有人，并返回它们所工作的城市。

查询计划

```
+----------------------+------+-------------------------+-------------------------+
| Operator             |Esti- | Variables               | Other                   |
|                      |mated |                         |                         |
|                      |Rows  |                         |                         |
+----------------------+------+-------------------------+-------------------------+
| +ProduceResults      |   1  | city.name               | city.name               |
| |                    +------+-------------------------+-------------------------+
| +Projection          |   1  | city.name -- anon[70],  | {city.name :city.name}  |
| |                    |      |  anon[93], city, fs, p  |                         |
| |                    +------+-------------------------+-------------------------+
| +OptionalExpand(All) |   1  | anon[93], city --       | (p)-[:WORKS_IN]->(city) |
| |                    |      |  anon[70], fs, p        |                         |
| |                    +------+-------------------------+-------------------------+
| +Filter              |   1  | anon[70], fs, p         | anon[70]                |
| |                    +------+-------------------------+-------------------------+
| +Projection          |   1  | anon[70] -- fs, p       | {p : p, fs : fs,: fs >  |
| |                    |      |                         |   {AUTOINT0}}           |
| |                    +------+-------------------------+-------------------------+
| +EagerAggregation    |   1  | fs -- p                 | p                       |
| |                    +------+-------------------------+-------------------------+
| +Expand(All)         |   2  | anon[17], f -- p        |(p)-[:FRIENDS_WITH]->(f) |
| |                    +------+-------------------------+-------------------------+
| +NodeByLabelScan     |  14  | p                       | :Person                 |
+----------------------+------+-------------------------+-------------------------+
Total database accesses: ?
```

3.7.3.2　SemiApply

测试一个模式断言的存在性。SemiApply 从它的子运算符中获取一行，并将其作为右端的叶节点运算符的输入。如果右端运算符至少产生一行结果，左端的这一行由 SemiApply 运算符产生。这使得 SemiApply 成为一个过滤运算符，可大量运用在查询的模式断言中。

查询

```
MATCH (p:Person)
WHERE (p)-[:FRIENDS_WITH]->()
RETURN p.name
```

查到有朋友的所有人。

查询计划

```
+-----------------+----------------+-------------------------+-----------------+
| Operator        | Estimated Rows | Variables               |           Other |
+-----------------+----------------+-------------------------+-----------------+
| +ProduceResults |             11 |                  p.name |          p.name |
| |               +----------------+-------------------------+-----------------+
| +Projection     |             11 |           p.name -- p   | {p.name : p.name} |
| |               +----------------+-------------------------+-----------------+
| +SemiApply      |             11 |                       p |                 |
```


仅供非商业用途或交流学习使用

```
| |\                +----------------+------------------------+------------------+
| | +Expand(All)    |              2 | anon[27], anon[45] -- p | (p)-[:FRIENDS_WITH]|
| | |               |                |                         |  ->()            |
| | |               +----------------+------------------------+------------------+
| | +Argument       |             14 |                      p |                  |
| |                 +----------------+------------------------+------------------+
| +NodeByLabelScan  |             14 |                      p |          :Person |
+-------------------+----------------+------------------------+------------------+
Total database accesses:?
```

3.7.3.3 AntiSemiApply

测试一个模式断言的存在性。AntiSemiApply 具有的功能与 SemiApply 相反，它进行反向过滤。

查询

```
MATCH (me:Person { name: "me" }),(other:Person)
WHERE NOT (me)-[:FRIENDS_WITH]->(other)
RETURN other.name
```

查找所有不是我朋友的人的名字。

查询计划

```
+--------------------+-----+-------------------------+--------------------------+
| Operator           |Esti-| Variables               | Other                    |
|                    |mated|                         |                          |
|                    |Rows |                         |                          |
+--------------------+-----+-------------------------+--------------------------+
| +ProduceResults    |   4 | other.name              | other.name               |
| |                  +-----+-------------------------+--------------------------+
| +Projection        |   4 | other.name -- me, other | {other.name : other.name}|
| |                  +-----+-------------------------+--------------------------+
| +AntiSemiApply     |   4 | me, other               |                          |
| |\                 +-----+-------------------------+--------------------------+
| | +Expand(Into)    |   0 | anon[62] -- me, other   | (me)-[:FRIENDS_WITH]->   |
| | |                |     |                         |   (other)                |
| | |                +-----+-------------------------+--------------------------+
| | +Argument        |  14 | me, other               |                          |
| |                  +-----+-------------------------+--------------------------+
| +CartesianProduct  |  14 | me--other               |                          |
| |\                 +-----+-------------------------+--------------------------+
| | +NodeByLabelScan |  14 | other                   | :Person                  |
| |                  +-----+-------------------------+--------------------------+
| +NodeIndexSeek     |   1 | me                      | :Person(name)            |
+--------------------+-----+-------------------------+--------------------------+
Total database accesses: ?
```

3.7.3.4 LetSemiApply

测试模式断言的存在性。当一个查询包含多个模式断言时，LetSemiApply 将用于处理它们中的第一个。它会记录断言的评估结果，但会留下过滤器到另外一个运算符。

查询

```
MATCH (other:Person)
WHERE (other)-[:FRIENDS_WITH]->() OR (other)-[:WORKS_IN]->()
RETURN other.name
```


仅供非商业用途或交流学习使用

找到有一个朋友或者在某地工作的所有人的名字。LetSemiApply 运算符将用于检查每个人的 FRIENDS_WITH 关系。

查询计划

```
+--------------------+-----+--------------------------+-----------------------+
| Operator           |Esti-| Variables                | Other                 |
|                    |mated|                          |                       |
|                    |Rows |                          |                       |
+--------------------+-----+--------------------------+-----------------------+
| +ProduceResults    |  13 | other.name               | other.name            |
| |                  +-----+--------------------------+-----------------------+
| +Projection        |  13 | other.name -- anon[27],  | {other.name:other.name}|
| |                  |     |  other                   |                       |
| |                  +-----+--------------------------+-----------------------+
| +SelectOrSemiApply |  13 | anon[27] -- other        | anon[27]              |
| |\                 +-----+--------------------------+-----------------------+
| | +Expand(All)     |  15 | anon[66], anon[80]--other | (other)-[:WORKS_IN]->()|
| | |                +-----+--------------------------+-----------------------+
| | +Argument        |  14 | other                    |                       |
| |                  +-----+--------------------------+-----------------------+
| +LetSemiApply      |  14 | anon[27] -- other        |                       |
| |\                 +-----+--------------------------+-----------------------+
| | +Expand(All)     |   2 | anon[35], anon[53]--other | (other)-[:FRIENDS_WITH]|
| | |                |     |                          |  ->()                 |
| | |                +-----+--------------------------+-----------------------+
| | +Argument        |  14 | other                    |                       |
| |                  +-----+--------------------------+-----------------------+
| +NodeByLabelScan   |  14 | other                    | :Person               |
+--------------------+-----+--------------------------+-----------------------+
Total database accesses: ?
```

3.7.3.5 LetAntiSemiApply

测试模式断言的存在性。当一个查询包含多个模式断言时，LetAntiSemiApply 将用于处理它们中的第一个。它会记录断言的评估结果，但会留下过滤器到另外一个运算符。下面的查询语句查找没有任何朋友或者在某地工作的所有人。

查询

```
MATCH (other:Person)
WHERE NOT ((other)-[:FRIENDS_WITH]->()) OR (other)-[:WORKS_IN]->()
RETURN other.name
```

查找没有任何朋友或者在某地工作的所有人。LetAntiSemiApply 运算符将用于检查每个人的 FRIENDS_WITH 关系的存在性。

查询计划

```
+--------------------+-----+----------------------+---------------------------+
| Operator           |Esti-| Variables            | Other                     |
|                    |mated|                      |                           |
|                    |Rows |                      |                           |
+--------------------+-----+----------------------+---------------------------+
| +ProduceResults    |  11 | other.name           | other.name                |
| |                  +-----+----------------------+---------------------------+
| +Projection        |  11 | other.name           | {other.name:other.name}   |
```


仅供非商业用途或交流学习使用

```
| |                    |      |  -- anon[31], other  |                              |
| |                    +------+----------------------+------------------------------+
| +SelectOrSemiApply   |  11  | anon[31] -- other    | anon[31]                     |
| |\                   +------+----------------------+------------------------------+
| | +Expand(All)       |  15  | anon[71], anon[85]   | (other)-[:WORKS_IN]->()      |
| | |                  |      |  -- other            |                              |
| | |                  +------+----------------------+------------------------------+
| | +Argument          |  14  | other                |                              |
| |                    +------+----------------------+------------------------------+
| +LetAntiSemiApply    |  14  | anon[31] -- other    |                              |
| |\                   +------+----------------------+------------------------------+
| | +Expand(All)       |   2  | anon[39], anon[57]   | (other)-[:FRIENDS_WITH]->()  |
| | |                  |      |  -- other            |                              |
| | |                  +------+----------------------+------------------------------+
| | +Argument          |  14  | other                |                              |
| |                    +------+----------------------+------------------------------+
| +NodeByLabelScan     |  14  | other                | :Person                      |
+----------------------+------+----------------------+------------------------------+
Total database accesses: ?
```

3.7.3.6 SelectOrSemiApply

测试一个模式断言的存在性并评估一个断言。这个运算符允许将一般的断言与检查存在性的断言放在一起。首先评估普通表达式，仅当它返回 false 时模式断言才会执行。

查询

```
MATCH (other:Person)
WHERE other.age > 25 OR (other)-[:FRIENDS_WITH]->()
RETURN other.name
```

查找有朋友或者年龄大于 25 的所有人的名字。

查询计划

```
+----------------------+------+----------------------+------------------------------+
| Operator             |Esti- | Variables            | Other                        |
|                      |mated |                      |                              |
|                      |Rows  |                      |                              |
+----------------------+------+----------------------+------------------------------+
| +ProduceResults      |  11  | other.name           | other.name                   |
| |                    +------+----------------------+------------------------------+
| +Projection          |  11  | other.name--other    | {other.name : other.name}    |
| |                    +------+----------------------+------------------------------+
| +SelectOrSemiApply   |  11  | other                | other.age > { AUTOINT0}      |
| |\                   +------+----------------------+------------------------------+
| | +Expand(All)       |   2  | anon[53], anon[71]   | (other)-[:FRIENDS_WITH]->()  |
| | |                  |      |  -- other            |                              |
| | |                  +------+----------------------+------------------------------+
| | +Argument          |  14  | other                |                              |
| |                    +------+----------------------+------------------------------+
| +NodeByLabelScan     |  14  | other                | :Person                      |
+----------------------+------+----------------------+------------------------------+
Total database accesses: ?
```

3.7.3.7 SelectOrAntiSemiApply

测试一个模式断言的存在性并评估一个断言。

仅供非商业用途或交流学习使用

查询

```
MATCH (other:Person)
WHERE other.age > 25 OR NOT (other)-[:FRIENDS_WITH]->()
RETURN other.name
```

查找没有朋友或者年龄大于 25 的所有人的名字。

查询计划

```
+-----------------------+-----+------------------+--------------------------+
| Operator              |Esti-| Variables        | Other                    |
|                       |mated|                  |                          |
|                       |Rows |                  |                          |
+-----------------------+-----+------------------+--------------------------+
| +ProduceResults       |   4 | other.name       | other.name               |
| |                     +-----+------------------+--------------------------+
| +Projection           |   4 | other.name--other |{other.name : other.name} |
| |                     +-----+------------------+--------------------------+
| +SelectOrAntiSemiApply|   4 | other            | other.age > other.name}  |
| |\                    +-----+------------------+--------------------------+
| | +Expand(All)        |   2 | anon[57],anon[75] |(other)-[:FRIENDS_WITH]->()|
| | |                   |     |  -- other        |                          |
| | |                   +-----+------------------+--------------------------+
| | +Argument           |  14 | other            |                          |
| |                     +-----+------------------+--------------------------+
| +NodeByLabelScan      |  14 | other            | :Person                  |
+-----------------------+-----+------------------+--------------------------+
Total database accesses: ?
```

3.7.3.8　ConditionalApply

检查一个变量是否不为 null，如果是，那么就执行右边的部分。

查询

```
MERGE (p:Person { name: 'Andres' })
ON MATCH SET p.exists = TRUE
```

查看是否存在一个名为'Andres'的人。如果找到，就设置他的 exists 属性为 true。

查询计划

```
+----------------------+----------------+-----------+---------------+
| Operator             | Estimated Rows | Variables |       Other   |
+----------------------+----------------+-----------+---------------+
| +ProduceResults      |              1 |           |               |
| |                    +----------------+-----------+---------------+
| +EmptyResult         |                |           |               |
| |                    +----------------+-----------+---------------+
| +AntiConditionalApply|              1 |         p |               |
| |\                   +----------------+-----------+---------------+
| | +MergeCreateNode   |              1 |         p |               |
| |                    +----------------+-----------+---------------+
| +ConditionalApply    |              1 |         p |               |
| |\                   +----------------+-----------+---------------+
| | +SetNodeProperty   |              1 |         p |               |
| | |                  +----------------+-----------+---------------+
| | +Argument          |              1 |         p |               |
```


仅供非商业用途或交流学习使用

```
| |                      +----------------+----------+---------------+
| +Optional              |              1 |        p |               |
| |                      +----------------+----------+---------------+
| +NodeIndexSeek         |              1 |        p | :Person(name) |
+------------------------+----------------+----------+---------------+
Total database accesses: ?
```

3.7.3.9 AntiConditionalApply

检查一个变量是否为 null，如果是，那么就执行右边的部分。

查询

```
MERGE (p:Person { name: 'Andres' })
ON CREATE SET p.exists = TRUE
```

查看是否存在一个名为'Andres'的人。如果没找到，则创建一个并设置其 exists 属性为 true。

查询计划

```
+------------------------+----------------+-----------+---------------+
| Operator               | Estimated Rows | Variables |       Other   |
+------------------------+----------------+-----------+---------------+
| +ProduceResults        |              1 |           |               |
| |                      +----------------+-----------+---------------+
| +EmptyResult           |                |           |               |
| |                      +----------------+-----------+---------------+
| +AntiConditionalApply  |              1 |         p |               |
| |\                     +----------------+-----------+---------------+
| | +SetNodeProperty     |              1 |         p |               |
| | |                    +----------------+-----------+---------------+
| | +MergeCreateNode     |              1 |         p |               |
| |                      +----------------+-----------+---------------+
| +Optional              |              1 |         p |               |
| |                      +----------------+-----------+---------------+
| +NodeIndexSeek         |              1 |         p | :Person(name) |
+------------------------+----------------+-----------+---------------+
Total database accesses: ?
```

3.7.3.10 AssertSameNode

这个运算符用于确保没有违背唯一性约束。

查询

```
MERGE (t:Team { name: 'Engineering', id: 42 })
```

查看 Team 中是否存在给定的名称和 id 的成员。如果不存在，则创建一个。如果存在:Team(name)和:Team(id)两个唯一性约束，通过 UniqueIndexSeek 找到的任何节点必定是同一个节点，否则就违背了唯一性约束。

查询计划

```
+--------------------------------+----------------+-----------+-------------+
| Operator                       | Estimated Rows | Variables |     Other   |
+--------------------------------+----------------+-----------+-------------+
| +ProduceResults                |              1 |           |             |
```


仅供非商业用途或交流学习使用

```
| |                                 +----------------+----------+-------------+
| +EmptyResult                      |                |          |             |
| |                                 +----------------+----------+-------------+
| +AntiConditionalApply             |              1 |        t |             |
| |\                                +----------------+----------+-------------+
| | +MergeCreateNode                |              1 |        t |             |
| |                                 +----------------+----------+-------------+
| +Optional                         |              1 |        t |             |
| |                                 +----------------+----------+-------------+
| +AssertSameNode                   |              0 |        t |             |
| |\                                +----------------+----------+-------------+
| | +NodeUniqueIndexSeek(Locking)   |              1 |        t | :Team(id)   |
| |                                 +----------------+----------+-------------+
| +NodeUniqueIndexSeek(Locking)     |              1 |        t | :Team(name) |
+-----------------------------------+----------------+----------+-------------+
Total database accesses: ?
```

3.7.3.11　NodeHashJoin

使用哈希表，NodeHashJoin 将来自左端和右端的输入连接起来。

查询

```
MATCH (andy:Person { name:'Andreas' })-[:WORKS_IN]->(loc)<-[:WORKS_IN]-
(matt:Person { name:'Mattis' })
RETURN loc.name
```

返回匹配到的两个人同时都在那里工作的地名。

查询计划

```
+------------------+-----+---------------------------+-------------------------+
| Operator         |Esti-| Variables                 | Other                   |
|                  |mated|                           |                         |
|                  |Rows |                           |                         |
+------------------+-----+---------------------------+-------------------------+
| +ProduceResults  |  10 | loc.name                  | loc.name                |
| |                +-----+---------------------------+-------------------------+
| +Projection      |  10 | loc.name -- anon[37],     | {loc.name :loc.name}    |
| |                |     |  anon[56], andy, loc, matt|                         |
| |                +-----+---------------------------+-------------------------+
| +Filter          |  10 | anon[37], anon[56], andy, | NOT(anon[37] ==anon[56])|
| |                |     |  loc, matt                |                         |
| |                +-----+---------------------------+-------------------------+
| +NodeHashJoin    |  10 | anon[37], andy -- anon[56],| loc                    |
| |                |     |   loc, matt               |                         |
| |\               +-----+---------------------------+-------------------------+
| | +Expand(All)   |  19 | anon[56], loc -- matt     |(matt)-[:WORKS_IN]->(loc)|
| | |              +-----+---------------------------+-------------------------+
| | +NodeIndexSeek |   1 | matt                      | :Person(name)           |
| |                +-----+---------------------------+-------------------------+
| +Expand(All)     |  19 | anon[37], loc -- andy     |(andy)-[:WORKS_IN]->(loc)|
| |                +-----+---------------------------+-------------------------+
| +NodeIndexSeek   |   1 | andy                      | :Person(name)           |
+------------------+-----+---------------------------+-------------------------+
Total database accesses: ?
```


仅供非商业用途或交流学习使用

3.7.3.12 三元（Triadic）

三元用于解决三元查询，如很常用的查找我朋友的朋友中那些还不是我朋友的人。它先将所有的朋友放入一个集合，然后再检查他们是否已经与我相连。

查询

```
MATCH (me:Person)-[:FRIENDS_WITH]-()-[:FRIENDS_WITH]-(other)
WHERE NOT (me)-[:FRIENDS_WITH]-(other)
RETURN other.name
```

查找我朋友的朋友中那些还不是我朋友的所有人。

查询计划

```
+------------------+------+-----------------------------+--------------------------+
| Operator         |Esti- | Variables                   | Other                    |
|                  |mated |                             |                          |
|                  |Rows  |                             |                          |
+------------------+------+-----------------------------+--------------------------+
| +ProduceResults  |   0  | other.name                  | other.name               |
| |                +------+-----------------------------+--------------------------+
| +Projection      |   0  | other.name--anon[18],       | {other.name: other.name} |
| |                |      |  anon[35],anon[37],me,other |                          |
| |                +------+-----------------------------+--------------------------+
| +TriadicSelection|   0  | anon[18], anon[35], me --   | me,anon[35], other       |
| |                |      |  anon[37], other            |                          |
| |\               +------+-----------------------------+--------------------------+
| | +Filter        |   0  | anon[37], other             | NOT(anon[18] == anon[37]) |
| | |              +------+-----------------------------+--------------------------+
| | +Expand(All)   |   0  | anon[37], other             |()-[:FRIENDS_WITH]-(other) |
| | |              +------+-----------------------------+--------------------------+
| | +Argument      |   4  |                             |                          |
| |                +------+-----------------------------+--------------------------+
| +Expand(All)     |   4  | anon[18], anon[35] -- me    | (me)-[:FRIENDS_WITH]-()  |
| |                +------+-----------------------------+--------------------------+
| +NodeByLabelScan |  14  | me                          | :Person                  |
+------------------+------+-----------------------------+--------------------------+
Total database accesses: ?
```

3.7.4 行运算符

这些运算符将其他运算符产生的行转换为一个新的行集合（Set）。

3.7.4.1 Eager

为了隔离的目的，这个运算符确保在继续之前将那些会影响后续操作的运算在整个数据集上被完全地执行。否则，它可能触发无限循环。当导入数据或者迁移图结构时，Eager 运算符会引起很高的内存消耗。在这种情况下，可将操作分解为更简单的步骤。例如，可以分别地导入节点和关系。另外，也可以先返回要更新的数据，然后再执行更新语句。

查询

```
MATCH (a)-[r]-(b)
DELETE r,a,b
```

```
MERGE ()
```

查询计划

```
+--------------------------+-----------+------+------+-----------------+-------+
| Operator                 | Estimated | Rows | DB   | Variables       | Other |
|                          | Rows      |      | Hits |                 |       |
+--------------------------+-----------+------+------+-----------------+-------+
| +ProduceResults          |         1 |    0 |    0 |                 |       |
| |                        +-----------+------+------+-----------------+-------+
| +EmptyResult             |           |    0 |    0 |                 |       |
| |                        +-----------+------+------+-----------------+-------+
| +Apply                   |         1 |  504 |    0 | a,b,r--anon[38] |       |
| |\                       +-----------+------+------+-----------------+-------+
| | +AntiConditionalApply  |         1 |  504 |    0 | anon[38]        |       |
| | |\                     +-----------+------+------+-----------------+-------+
| | | +MergeCreateNode     |         1 |    0 |    0 | anon[38]        |       |
| | |                      +-----------+------+------+-----------------+-------+
| | +Optional              |        35 |  504 |    0 | anon[38]        |       |
| | |                      +-----------+------+------+-----------------+-------+
| | +AllNodesScan          |        35 |  504 |  540 | anon[38]        |       |
| |                        +-----------+------+------+-----------------+-------+
| +Eager                   |           |   36 |    0 | a, b, r         |       |
| |                        +-----------+------+------+-----------------+-------+
| +Delete(3)               |        36 |   36 |   39 | a, b, r         |       |
| |                        +-----------+------+------+-----------------+-------+
| +Eager                   |           |   36 |    0 | a, b, r         |       |
| |                        +-----------+------+------+-----------------+-------+
| +Expand(All)             |        36 |   36 |   71 | a, r -- b       |(b)-[r:]-(a)|
| |                        +-----------+------+------+-----------------+-------+
| +AllNodesScan            |        35 |   35 |   36 | b               |       |
+--------------------------+-----------+------+------+-----------------+-------+
Total database accesses: 686
```

3.7.4.2　Distinct

移除输入行流中重复的行。

查询

```
MATCH (l:Location)<-[:WORKS_IN]-(p:Person)
RETURN DISTINCT l
```

查询计划

```
+------------------+-----+------+------+----------------+----------------------+
| Operator         |Esti-| Rows | DB   | Variables      | Other                |
|                  |mated|      | Hits |                |                      |
|                  |Rows |      |      |                |                      |
+------------------+-----+------+------+----------------+----------------------+
| +ProduceResults  |  14 |    6 |    0 | l              | l                    |
| |                +-----+------+------+----------------+----------------------+
| +Distinct        |  14 |    6 |    0 | l              | l                    |
| |                +-----+------+------+----------------+----------------------+
| +Filter          |  15 |   15 |   15 | anon[19], l, p | p:Person             |
| ||               +-----+------+------+----------------+----------------------+
| +Expand(All)     |  15 |   15 |   25 | anon[19], p--l | (l)<-[:WORKS_IN]-(p) |
| |                +-----+------+------+----------------+----------------------+
| +NodeByLabelScan |  10 |   10 |   11 | l              | :Location            |
+------------------+-----+------+------+----------------+----------------------+
Total database accesses: 51
```

3.7.4.3 Eager 聚合

即时加载潜在的结果并存入哈希 map 中，使用分组键作为 map 的键。

查询

```
MATCH (l:Location)<-[:WORKS_IN]-(p:Person)
RETURN l.name AS location, collect(p.name) AS people
```

查询计划

```
+------------------+-----+------+------+-----------------+----------------------+
| Operator         |Esti-|Rows  | DB   | Variables       | Other                |
|                  |mated|      | Hits |                 |                      |
|                  |Rows |      |      |                 |                      |
+------------------+-----+------+------+-----------------+----------------------+
| +ProduceResults  |  4  |   6  |   0  | location, people | location, people     |
| |                +-----+------+------+-----------------+----------------------+
| +EagerAggregation|  4  |   6  |  15  | people--location | location             |
| |                +-----+------+------+-----------------+----------------------+
| +Projection      | 15  |  15  |  15  | location --      | {location : l.name,  |
| |                |     |      |      |  anon[19], l, p  |  p: p}               |
| |                +-----+------+------+-----------------+----------------------+
| +Filter          | 15  |  15  |  15  | anon[19], l, p   | p:Person             |
| |                +-----+------+------+-----------------+----------------------+
| +Expand(All)     | 15  |  15  |  25  | anon[19], p -- l | (l)<-[:WORKS_IN]-(p) |
| |                +-----+------+------+-----------------+----------------------+
| +NodeByLabelScan | 10  |  10  |  11  | l                | :Location            |
+------------------+-----+------+------+-----------------+----------------------+
Total database accesses: 81
```

3.7.4.4 从计数库获取节点数量

从计数库中得到节点的数量，比通过计数方式的 Eager 聚合要快。然而，计数库中只保存了有限范围的组合，因此 Eager 聚合对很多复杂的查询依然很有用。例如，可以从计数库中得到所有节点和拥有某个标签的节点的数量，但无法获取到超过一个标签的节点的数量。

查询

```
MATCH (p:Person)
RETURN count(p) AS people
```

结果

```
+--------------------------+-----+------+------+-----------+---------------------+
| Operator                 |Esti-| Rows |  DB  | Variables | Other               |
|                          |mated|      | Hits |           |                     |
|                          |Rows |      |      |           |                     |
+--------------------------+-----+------+------+-----------+---------------------+
| +ProduceResults          |  4  |   1  |   0  | people    | people              |
| |                        +-----+------+------+-----------+---------------------+
| +NodeCountFromCountStore |  4  |   1  |   0  |  people   | count( (:Person) )  |
|                          |     |      |      |           |  AS people          |
+--------------------------+-----+------+------+-----------+---------------------+
Total database accesses: 0
```

3.7.4.5　从计数库获取关系数量

从计数库中得到关系的数量，比通过计数方式的 Eager 聚合要快。然而，计数库中只保存了有限范围的组合，因此 Eager 聚合对很多复杂的查询依然很有用。例如，可以从计数库中得到所有关系，某个类型的关系的数量以及末尾节点上拥有某个标签的关系的数量，但无法获取到两端节点都有标签的关系的数量。

查询

```
MATCH (p:Person)-[r:WORKS_IN]->()
RETURN count(r) AS jobs
```

查询计划

```
+--------------------+------+------+------+-----------+---------------------------+
| Operator           |Esti- | Rows | DB   | Variables | Other                     |
|                    |mated |      | Hits |           |                           |
|                    |Rows  |      |      |           |                           |
+--------------------+------+------+------+-----------+---------------------------+
| +ProduceResults    |    4 |    1 |    0 |      jobs | jobs                      |
| |                  +------+------+------+-----------+---------------------------+
| +RelationshipCount |    4 |    1 |    1 |      jobs | count( (:Person)-         |
|  FromCountStore    |      |      |      |           |   [:WORKS_IN]->() ) AS jobs|
+--------------------+------+------+------+-----------+---------------------------+
Total database accesses: 1
```

3.7.4.6　过滤

过滤来自子运算符的每一行，仅仅让断言为 true 的结果通过。

查询

```
MATCH (p:Person)
WHERE p.name =~ '^a.*'
RETURN p
```

查询计划

```
+-----------------+------+------+------+------------+---------------------------+
| Operator        |Esti- | Rows | DB   | Variables  | Other                     |
|                 |mated |      | Hits |            |                           |
|                 |Rows  |      |      |            |                           |
+-----------------+------+------+------+------------+---------------------------+
| +ProduceResults |   14 |    0 |    0 | p          | p                         |
| |               +------+------+------+------------+---------------------------+
| +Filter         |   14 |    0 |   14 | p          | p.name ~= /{ AUTOSTRING0}/|
| |               +------+------+------+------------+---------------------------+
| +NodeIndexScan  |   14 |   14 |   15 | p          | :Person(name)             |
+-----------------+------+------+------+------------+---------------------------+
Total database accesses: 29
```

3.7.4.7　Limit

返回输入的前 n 行。

查询

```
MATCH (p:Person)
RETURN p
LIMIT 3
```

查询计划

```
+-----------------+----------------+------+---------+-----------+------------+
| Operator        | Estimated Rows | Rows | DB Hits | Variables |      Other |
+-----------------+----------------+------+---------+-----------+------------+
| +ProduceResults |              3 |    3 |       0 |         p |          p |
| |               +----------------+------+---------+-----------+------------+
| +Limit          |              3 |    3 |       0 |         p | Literal(3) |
| |               +----------------+------+---------+-----------+------------+
| +NodeByLabelScan |            14 |    3 |       4 |         p |    :Person |
+-----------------+----------------+------+---------+-----------+------------+
Total database accesses: 4
```

3.7.4.8 Projection

对于输入的每一行，projection 将评估表达式并产生一行表达式的结果。

查询

```
RETURN 'hello' AS greeting
```

查询计划

```
+-----------------+----------------+------+---------+-----------+-----------------+
| Operator        | Estimated Rows | Rows | DB Hits | Variables |           Other |
+-----------------+----------------+------+---------+-----------+-----------------+
| +ProduceResults |              1 |    1 |       0 |  greeting |        greeting |
| |               +----------------+------+---------+-----------+-----------------+
| +Projection     |              1 |    1 |       0 |  greeting | {greeting :     |
|                 |                |      |         |           | {AUTOSTRING0}}  |
+-----------------+----------------+------+---------+-----------+-----------------+
Total database accesses: 0
```

3.7.4.9 Skip

跳过输入行的前 n 行。

查询

```
MATCH (p:Person)
RETURN p
ORDER BY p.id
SKIP 1
```

查询计划

```
+-----------------+-----+------+------+----------------------+-------------+
| Operator        |Esti-| Rows | DB   | Variables            | Other       |
|                 |mated|      | Hits |                      |             |
|                 |Rows |      |      |                      |             |
+-----------------+-----+------+------+----------------------+-------------+
| +ProduceResults |  14 |   13 |    0 | p                    | p           |
| |               +-----+------+------+----------------------+-------------+
```

```
| +Projection      | 14  |  13  |   0  | p--anon[35],anon[59],p| {p : }       |
| |                +-----+------+------+-----------------------+--------------+
| +Skip            | 14  |  13  |   0  | anon[35], anon[59], p | { AUTOINT0} |
| |                +-----+------+------+-----------------------+--------------+
| +Sort            | 14  |  14  |   0  | anon[35], anon[59], p | anon[59]     |
| |                +-----+------+------+-----------------------+--------------+
| +Projection      | 14  |  14  |  14  | anon[59]--anon[35], p | { : , : .id}|
| |                +-----+------+------+-----------------------+--------------+
| +Projection      | 14  |  14  |   0  | anon[35] -- p         | { : p@7}     |
| |                +-----+------+------+-----------------------+--------------+
| +NodeByLabelScan| 14  |  14  |  15  | p                     | :Person      |
+------------------+-----+------+------+-----------------------+--------------+
Total database accesses: 29
```

3.7.4.10　Sort

根据给定的键进行排序。

查询

```
MATCH (p:Person)
RETURN p
ORDER BY p.name
```

查询计划

```
+-----------------+-----+------+------+-------------------------+----------------+
| Operator        |Esti-| Rows | DB   | Variables               | Other          |
|                 |mated|      | Hits |                         |                |
|                 |Rows |      |      |                         |                |
+-----------------+-----+------+------+-------------------------+----------------+
| +ProduceResults | 14  |  14  |   0  | p                       | p              |
| |               +-----+------+------+-------------------------+----------------+
| +Projection     | 14  |  14  |   0  | p--anon[24],anon[37], p | {p : }         |
| |               +-----+------+------+-------------------------+----------------+
| +Sort           | 14  |  14  |   0  | anon[24], anon[37], p   | anon[37]       |
| |               +-----+------+------+-------------------------+----------------+
| +Projection     | 14  |  14  |  14  | anon[37] -- anon[24], p | { : , : .name} |
| |               +-----+------+------+-------------------------+----------------+
| +Projection     | 14  |  14  |   0  | anon[24] -- p           | { : p@7}       |
| |               +-----+------+------+-------------------------+----------------+
| +NodeByLabelScan| 14  |  14  |  15  | p                       | :Person        |
+-----------------+-----+------+------+-------------------------+----------------+
Total database accesses: 29
```

3.7.4.11　Top

返回根据给定键排序后的前 n 行。实际的运算符是 Top，它仅保留前 X 行，而不像排序需要作用于整个输入。

查询

```
MATCH (p:Person)
RETURN p
ORDER BY p.name
LIMIT 2
```

查询计划

```
+-----------------+-----+------+------+-----------------------+-----------------+
| Operator        |Esti-| Rows | DB   | Variables             | Other           |
|                 |mated|      | Hits |                       |                 |
|                 |Rows |      |      |                       |                 |
+-----------------+-----+------+------+-----------------------+-----------------+
| +ProduceResults |   2 |    2 |    0 | p                     | p               |
| |               +-----+------+------+-----------------------+-----------------+
| +Projection     |   2 |    2 |    0 | p--anon[24],anon[37],p | {p : }          |
| |               +-----+------+------+-----------------------+-----------------+
| +Top            |   2 |    2 |    0 | anon[24], anon[37], p |Literal(2);anon[37]|
| |               +-----+------+------+-----------------------+-----------------+
| +Projection     |  14 |   14 |   14 | anon[37]--anon[24], p | { : , : .name}  |
| |               +-----+------+------+-----------------------+-----------------+
| +Projection     |  14 |   14 |    0 | anon[24] -- p         | { : p@7}        |
| |               +-----+------+------+-----------------------+-----------------+
| +NodeByLabelScan|  14 |   14 |   15 | p                     | :Person         |
+-----------------+-----+------+------+-----------------------+-----------------+
Total database accesses: 29
```

3.7.4.12 Union

Union 将左右两个计划的结果连接在一起。

查询

```
MATCH (p:Location)
RETURN p.name
UNION ALL MATCH (p:Country)
RETURN p.name
```

查询计划

```
+--------------------+----------------+------+---------+------------+------------+
| Operator           | Estimated Rows | Rows | DB Hits | Variables  |      Other |
+--------------------+----------------+------+---------+------------+------------+
| +ProduceResults    |             10 |   11 |       0 | p.name     | p.name     |
| |                  +----------------+------+---------+------------+------------+
| +Union             |             10 |   11 |       0 | p.name     |            |
| |\                 +----------------+------+---------+------------+------------+
| | +Projection      |              1 |    1 |       1 | p.name -- p | {p.name :  |
| | |                |                |      |         |            |  p@83.name}|
| | |                +----------------+------+---------+------------+------------+
| | +NodeByLabelScan |              1 |    1 |       2 | p          | :Country   |
| |                  +----------------+------+---------+------------+------------+
| +Projection        |             10 |   10 |      10 | p.name -- p | {p.name :  |
| |                  |                |      |         |            |  p@7.name} |
| |                  +----------------+------+---------+------------+------------+
| +NodeByLabelScan   |             10 |   10 |      11 | p          | :Location  |
+--------------------+----------------+------+---------+------------+------------+
Total database accesses: 24
```

3.7.4.13 Unwind

将列表中的值以每行一个元素的形式返回。

查询

```
UNWIND range(1, 5) AS value
RETURN value;
```

查询计划

```
+-----------------+----------------+------+---------+-----------+-------+
| Operator        | Estimated Rows | Rows | DB Hits | Variables | Other |
+-----------------+----------------+------+---------+-----------+-------+
| +ProduceResults |             10 |    5 |       0 |     value | value |
| |               +----------------+------+---------+-----------+-------+
| +Unwind         |             10 |    5 |       0 |     value |       |
| |               +----------------+------+---------+-----------+-------+
| +EmptyRow       |              1 |    1 |       0 |           |       |
+-----------------+----------------+------+---------+-----------+-------+
Total database accesses: 0
```

3.7.4.14　调用过程

返回以 name 为序的所有标签。

查询

```
CALL db.labels() YIELD label
RETURN *
ORDER BY label
```

查询计划

```
+-----------------+----------------+------+---------+-----------+-----------------+
| Operator        | Estimated Rows | Rows | DB Hits | Variables |           Other |
+-----------------+----------------+------+---------+-----------+-----------------+
| +ProduceResults |          10000 |    4 |       0 |     label |           label |
| |               +----------------+------+---------+-----------+-----------------+
| +Sort           |          10000 |    4 |       0 |     label |           label |
| |               +----------------+------+---------+-----------+-----------------+
| +ProcedureCall  |          10000 |    4 |       1 |     label | db.labels() :: |
| |               |                |      |         |           |(label ::String) |
| |               +----------------+------+---------+-----------+-----------------+
| +EmptyRow       |              1 |    1 |       0 |           |                 |
+-----------------+----------------+------+---------+-----------+-----------------+
Total database accesses: 1
```

3.7.5　更新运算符

这些运算符用于在查询中更新图。

3.7.5.1　约束操作

在一对标签和属性上创建一个约束。下面的查询在带有 Country 标签节点的 name 属性上创建一个唯一性约束。

查询

```
CREATE CONSTRAINT ON (c:Country) ASSERT c.name IS UNIQUE
```

查询计划

```
+--------------------------+
| Operator                 |
+--------------------------+
| +CreateUniqueConstraint |
+--------------------------+
Total database accesses: ?
```

3.7.5.2 EmptyResult

即时加载产生的所有结果到 EmptyResult 运算符并丢弃掉。

查询

```
CREATE (:Person)
```

查询计划

```
+-----------------+----------------+------+---------+-----------+
| Operator        | Estimated Rows | Rows | DB Hits | Variables |
+-----------------+----------------+------+---------+-----------+
| +ProduceResults |              1 |    0 |       0 |           |
| |               +----------------+------+---------+-----------+
| +EmptyResult    |                |    0 |       0 |           |
| |               +----------------+------+---------+-----------+
| +CreateNode     |              1 |    1 |       2 | anon[8]   |
+-----------------+----------------+------+---------+-----------+
Total database accesses: 2
```

3.7.5.3 更新图

对图进行更新操作。

查询

```
CYPHER planner=rule
CREATE (:Person { name: 'Alistair' })
```

查询计划

```
+--------------+------+---------+-----------+------------+
| Operator     | Rows | DB Hits | Variables | Other      |
+--------------+------+---------+-----------+------------+
| +EmptyResult |    0 |       0 |           |            |
| |            +------+---------+-----------+------------+
| +UpdateGraph |    1 |       4 | anon[7]   | CreateNode |
+--------------+------+---------+-----------+------------+
Total database accesses: 4
```

3.7.5.4 Merge Into

当开始和结束节点都已经找到，Merge Into 用于找到这两个节点之间的所有关系或者创建一个新的关系。

查询

```
CYPHER planner=rule
MATCH (p:Person { name: 'me' }),(f:Person { name: 'Andres' })
```

```
MERGE (p)-[:FRIENDS_WITH]->(f)
```

查询计划

```
+--------------+------+---------+-----------------+-------------------------------+
| Operator     | Rows | DB Hits |      Variables  | Other                         |
+--------------+------+---------+-----------------+-------------------------------+
| +EmptyResult |   0  |     0   |                 |                               |
| |            +------+---------+-----------------+-------------------------------+
| +Merge(Into) |   1  |     5   | anon[68] -- f, p | (p)-[:FRIENDS_WITH]->(f)      |
| |            +------+---------+-----------------+-------------------------------+
| +SchemaIndex |   1  |     2   | f -- p          | { AUTOSTRING1}; :Person(name)|
| |            +------+---------+-----------------+-------------------------------+
| +SchemaIndex |   1  |     2   |           p     | { AUTOSTRING0}; :Person(name)|
+--------------+------+---------+-----------------+-------------------------------+
Total database accesses: 9
```

3.7.6　最短路径规划

本小节讲解 Cypher 中的最短路径查找是如何规划的。

不同的断言在规划最短路径时可能导致 Cypher 中产生不同的查询计划。如果断言可以在搜索路径时处理，Neo4j 将使用快速的双向广度优先搜索算法。因此，当路径上是普通断言时，这个快速算法可以一直确定地返回正确的结果。例如，当搜索有 Person 标签的所有节点的最短路径时，或者没有带有 name 属性的节点时。

如果在决定哪条路径是有效或者无效之前，断言需要检查所有路径，那么这个算法就不能可靠地找到最短路径。Neo4j 可能需要求助于比较慢的穷举深度优先搜索算法去寻找路径。

这两种算法的运行时间可能有数量级的差异，因此，对于时间敏感性查询确保使用的是快速算法很重要。

3.7.6.1　用快速算法检索最短路径

查询

```
MATCH (ms:Person { name: 'Martin Sheen' }),(cs:Person { name: 'Charlie
Sheen' }), p = shortestPath((ms)-[rels:ACTED_IN*]-(cs))
WHERE ALL (r IN rels WHERE exists(r.role))
RETURN p
```

这个查询可以使用快速算法——因为没有断言需要查看所有路径。

查询计划

```
+-----------------+-----+----+----+-----------------+-------------------------+
| Operator        |Esti-|Rows|DB  | Variables       | Other                   |
|                 |mated|    |Hits|                 |                         |
|                 |Rows |    |    |                 |                         |
+-----------------+-----+----+----+-----------------+-------------------------+
| +ProduceResults |  1  | 1  | 0  | p               | p                       |
| |               +-----+----+----+-----------------+-------------------------+
| +ShortestPath   |  1  | 1  | 9  | p, rels -- cs, ms | {p0 : all(r in rels   |
| |               |     |    |    |                 |  wherehasProp(r.role))} |
```

```
| |                  +-----+----+----+------------------+--------------------------+
| +CartesianProduct| 1   | 1  | 0  | ms -- cs         |                          |
| |\                 +-----+----+----+------------------+--------------------------+
| | +NodeIndexSeek | 1   | 1  | 2  | cs               | :Person(name)            |
| |                  +-----+----+----+------------------+--------------------------+
| +NodeIndexSeek   | 1   | 1  | 2  | ms               | :Person(name)            |
+------------------+-----+----+----+------------------+--------------------------+
Total database accesses: 13
```

3.7.6.2 需要检查路径上额外断言的最短路径规划

1. 考虑使用穷举搜索

在决定哪条是最短的匹配路径之前，WHERE 语句中的断言需要应用于最短路径模式。

查询

```
MATCH (cs:Person { name: 'Charlie Sheen' }),(ms:Person { name: 'Martin
Sheen' }), p = shortestPath((cs)-[*]-(ms))
WHERE length(p)> 1
RETURN p
```

与前面那个相反，这个查询在知道哪条是最短路径之前，需要检查所有的路径。因此，查询计划将使用慢一些的穷举搜索算法。

查询计划

```
+--------------------------+-----+----+----+------------------+------------------+
| Operator                 |Esti-|Rows|DB  | Variables        | Other            |
|                          |mated|    |Hits|                  |                  |
|                          |Rows |    |    |                  |                  |
+--------------------------+-----+----+----+------------------+------------------+
| +ProduceResults          | 0   | 1  | 0  | p                | p                |
| |                        +-----+----+----+------------------+------------------+
| +AntiConditionalApply    | 0   | 1  | 0  | cs -- anon[95],  |                  |
| |                        |     |    |    |  anon[113], ms, p|                  |
| |\                       +-----+----+----+------------------+------------------+
| | +Top1                  | 0   | 0  | 0  |anon[95],anon[113]| anon[95]         |
| | |                      |     |    |    |  ,ms, p          |                  |
| | |                      +-----+----+----+------------------+------------------+
| | +Projection            | 0   | 0  | 0  | anon[95] --      | { :length(p)}    |
| | |                      |     |    |    |  anon[113],ms,p  |                  |
| | |                      +-----+----+----+------------------+------------------+
| | +Filter                | 0   | 0  | 0  | anon[113], ms, p | length(p) >      |
| | |                      |     |    |    |                  |  {AUTOINT2}      |
| | |                      +-----+----+----+------------------+------------------+
| | +Projection            | 0   | 0  | 0  | p--anon[113], ms |{p:ProjectedPath(Set |
| | |                      |     |    |    |                  |(, cs),<function2>)} |
| | |                      +-----+----+----+------------------+------------------+
| | +VarLengthExpand(Into) | 0   | 0  | 0  | anon[113], ms    | (cs)-[:*]-(ms)   |
| | |                      +-----+----+----+------------------+------------------+
| | +Argument              | 1   | 0  | 0  |                  |                  |
| |                        +-----+----+----+------------------+------------------+
| +Apply                   | 1   | 1  | 0  | cs, ms--anon[113],|                 |
| |                        |     |    |    |  p               |                  |
| |\                       +-----+----+----+------------------+------------------+
| | +Optional              | 1   | 1  | 0  | anon[113], p     |                  |
```

```
| | |                          +-----+----+----+------------------+------------------+
| | +ShortestPath              |  0  | 1  | 1  | anon[113], p     | {p0 :length(p) > |
| | |                          |     |    |    |                  |  {AUTOINT2}}     |
| | |                          +-----+----+----+------------------+------------------+
| | +Argument                  |  1  | 1  | 0  |                  |                  |
| |                            +-----+----+----+------------------+------------------+
| +CartesianProduct            |  1  | 1  | 0  | cs -- ms         |                  |
| |\                           +-----+----+----+------------------+------------------+
| | +NodeIndexSeek             |  1  | 1  | 2  | ms               |:Person(name)     |
| |                            +-----+----+----+------------------+------------------+
| +NodeIndexSeek               |  1  | 1  | 2  | cs               |:Person(name)     |
+------------------------------+-----+----+----+------------------+------------------+
Total database accesses: 5
```

这种更费时的穷举查询计划使用 Apply/Optional 来确保，当快速算法无法找到结果的时候返回一个 null 结果，而不是简单地停止结果流。在查询计划的顶部，查询器使用了一个 AntiConditionalApply，如果路径变量指向的是 null，那么它将运行穷举搜索。

2. 禁止使用穷举搜索算法

这个查询与上面的查询一样，在知道哪条路径是最短路径之前需要检查所有路径。然而，使用 WITH 语句将使得查询计划不使用穷举搜索算法。由快速算法找到的任何路径接下来将被过滤掉，这可能会导致没有结果返回。

查询

```
MATCH (cs:Person { name: 'Charlie Sheen' }),(ms:Person { name: 'Martin
Sheen' }), p = shortestPath((cs)-[*]-(ms))
WITH p
WHERE length(p)> 1
RETURN p
```

查询计划

```
+--------------------+-----+----+----+----------------------+----------------------+
| Operator           |Esti-|Rows|DB  | Variables            | Other                |
|                    |mated|    |Hits|                      |                      |
|                    |Rows |    |    |                      |                      |
+--------------------+-----+----+----+----------------------+----------------------+
| +ProduceResults    |  1  | 1  | 0  | p                    | p                    |
| |                  +-----+----+----+----------------------+----------------------+
| +Filter            |  1  | 1  | 0  | anon[146], anon[113],| anon[146]            |
| |                  |     |    |    |  cs, ms, p           |                      |
| |                  +-----+----+----+----------------------+----------------------+
| +Projection        |  1  | 1  | 0  | anon[146]--anon[113],| {p : p, :length(p) > |
| |                  |     |    |    |  cs, ms, p           |  {AUTOINT2}}         |
| |                  +-----+----+----+----------------------+----------------------+
| +ShortestPath      |  1  | 1  | 1  | anon[113], p--cs, ms | {}                   |
| |                  +-----+----+----+----------------------+----------------------+
| +CartesianProduct  |  1  | 1  | 0  | cs -- ms             |                      |
| |\                 +-----+----+----+----------------------+----------------------+
| | +NodeIndexSeek   |  1  | 1  | 2  | ms                   | :Person(name)        |
| |                  +-----+----+----+----------------------+----------------------+
| +NodeIndexSeek     |  1  | 1  | 2  | cs                   | :Person(name)        |
+--------------------+-----+----+----+----------------------+----------------------+
Total database accesses: 5
```

第 4 章

◂Neo4j 程序开发▸

本章包括以下内容：

- Neo4j 开发入门
- Java API 嵌入式开发模式
- 各语言驱动包开发模式
- Neo4j HTTP API
- 其他开发技术介绍

本章内容将详细讨论如何将 Neo4j 与开发平台、编程语言相互集成。Neo4j 正式支持.Net、Java、JavaScript、Ruby、PHP 和 Python 的二进制 Bolt 协议驱动程序，这些开发平台通过引入相应的驱动程序包便可与 Neo4j 相互集成，然后就可以对 Neo4j 进行数据操作。另外，Neo4j 社区贡献者已经为大部分非主流编程语言编写了驱动程序，开发人员可以直接调用相应的 API。

本章所有实例代码都可以从 Neo4j 的 GitHub 主页找到：https://github.com/neo4j-examples，建议读者结合实例代码学习以达到事半功倍的效果。

4.1 Neo4j 开发入门

4.1.1 Java 嵌入式开发模式

读者大概已经了解到，Neo4j 是用 Java 语言开发、基于 JVM（Java Virtual Machine，Java 虚拟机）的一款产品，因此在 Neo4j 第一版发布之初，它是专门针对 Java 领域的，所以它能够与 Java 开发天然结合，Java 开发人员完全可以直接在代码中调用 Neo4j 的 API，并将对 Neo4j 数据库的操作嵌入在 Java 代码中，这就是嵌入式开发模式。

4.1.2 各语言驱动包开发模式

随着 Neo4j 不断改进和被各大应用领域的认可，其他开发语言如.Net、JavaScript、Python、PHP 等也希望能够和 Neo4j 相互集成，就像操作传统的关系数据库（如 SQL

Server、Oracle、MySQL）那样熟练地操作 Neo4j 这个图数据库。由此，驱动模式就被引入到 Neo4j 开发中。Neo4j 本质上是一款基于 JVM 的产品，最初，这意味着任何基于 JVM 的开发平台、语言才能够操作 Neo4j 数据库，但驱动模式被引入到 Neo4j 开发的主要原因就是让非基于 JVM 的开发平台、编程语言也能够操作 Neo4j 数据库。在驱动模式下，其他开发平台、语言通过它们专门基于 HTTP 的 HTTP API 的驱动包或驱动库就可以与 Neo4j 相互对话。

下面给出两种模式的结构图，如图 4-1 所示。

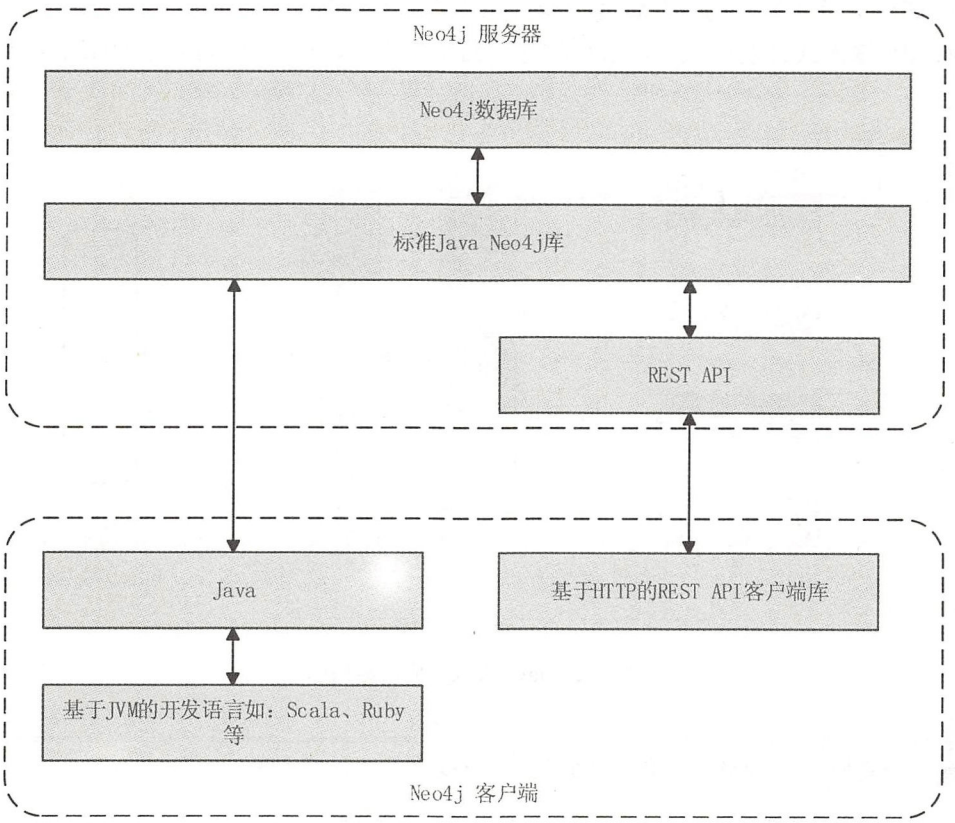


图 4-1　Neo4j 开发模式结构图

4.2 Java API 嵌入式开发模式

由于 Neo4j 是一款基于 JVM 的产品，可以直接在 Java 应用程序中使用 Neo4j 的 Java API 来进行编程开发，而 Java API 在应用程序和 Neo4j 的物理数据存储之间成为一种传送数据的机制。这样应用程序就可以使用 Neo4j 数据库的大部分功能，并能直接管理 Neo4j 数据库，例如所有 Neo4j 的操作逻辑、遍历、查询等操作都可以通过 Java 程序实现。从应用程序的角度看这是非常有用的。现在，我们只要知道通过 Java API 嵌入式开发模式，Java 应用程序和 Neo4j 将运行在同一个 JVM 上就足以满足需要了。

除了可以使用面向图数据库的 API 来操作节点、关系和路径对象，它还提供可定制的高速遍历和图算法。

作为一个 Java 开发人员，当在 Neo4j 中插入大量数据时可以充分体会到它的优势。利用非事务性、高并发批量插入 API，可以摄取数十亿的节点和关系并非常容易地将其存入 Neo4j 中并进行其他处理。这也满足我们对数据库的高性能需求。

另外，由于其灵活性和高性能，我们还可以使用 Java API 扩展 Neo4j 的功能来定制自己的数据库插件，也就是后续章节将要讨论的用户自定义过程。

Java API 嵌入式开发模式中，应用程序、Java API、Neo4j 数据的关系如图 4-2 所示。

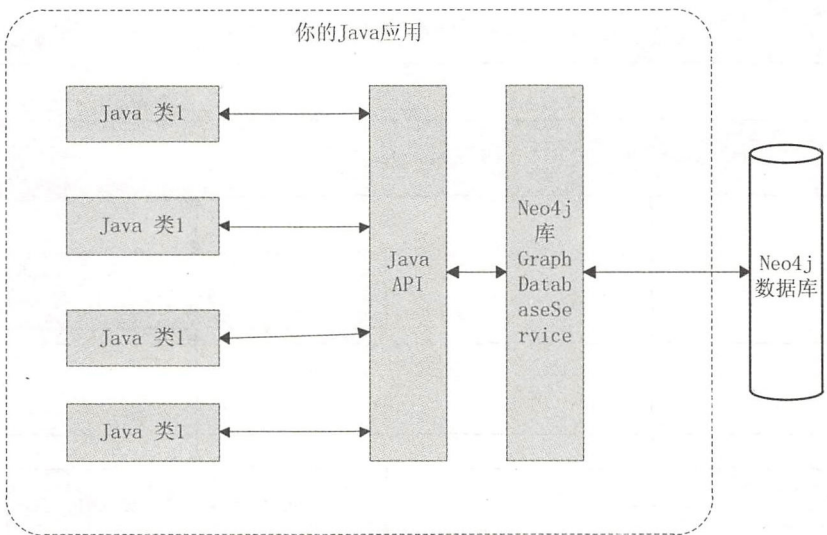


图 4-2　Java 嵌入式应用结构图



当在同一个 JVM 中运行自己的代码和 Neo4j 时，需要注意：

（1）不要创建或保留过多不需要的对象。大型缓存会将不需要的对象推向垃圾回收区域，因此增加了对垃圾回收的负担。

（2）尽量不要使用内部 Neo4j API。它们是供 Neo4j 内部调用，如有更改可能会破坏或更改 Neo4j 的固有特性。

（3）在嵌入式模式下运行时，不要启用-XX：+TrustFinalNonStaticFields JVM 标志。

4.2.1　Java 开发前的准备工作

这一小节的目的是做好开发前的准备工作，包括 JAR 包的引入、不同 IDE 下的引入方式和程序主体代码的编写。

4.2.1.1　引入 JAR 包

Java API 嵌入式开发模式需要引入指定的库文件（JAR 包），将 JAR 包绑定到程序中就

可以使用 Neo4j 的 Java 库了。

首先根据应用程序使用的 Java 版本选择合适的 Neo4j JAR 包版本，然后在应用程序项目中引入此 JAR 包，这样 Neo4j API 就嵌入到 Java 应用程序中。以下部分将介绍通过两个途径来引入 Neo4j JAR 包：

（1）将 JAR 文件直接添加到 Java 的执行路径“ClassPath”中。

（2）使用依赖关系管理来添加 Neo4j JAR 包的依赖。

1. 各版本的依赖库及其名称

由于 Neo4j 分为社区版和企业版。首先需要确定对应的版本和名称，表 4-1 概述了用于依赖管理工具的可用版本及其名称。

表 4-1　各版本的依赖库及其名称

版本	依赖库	描述	许可协议
Neo4j 社区版	org.neo4j:neo4j	一个高性能，有完全 ACID 事务的图数据库	GPLv3
Neo4j 企业版	org.neo4j:neo4j-enterprise	添加高级监控、在线备份和高可用性功能集群	AGPLv3

将 JAR 包直接添加到 Java 的执行路径里。

2. 获取 JAR 文件

在 Neo4j 解压版本中找到 neo4j-community\lib 文件夹，其中的 JAR 文件就是我们所需要的。

直接到 Maven Central Repository 中搜索:"org.neo4j"，也可以下载相关的 JAR 文件。

3. JDK 环境变量

可以从获取到的所有 JAR 文件中找到需要的包，然后将包所在的文件夹路径添加到 Java 环境路径-classpath 中即可。

4. Eclipse

右击工程，然后到 Build Path\Configure Build Path 对话框，在对话框中选择 Add External JARs，然后在浏览窗口中找到 neo4j-community\lib 文件夹并选择所有 JAR 文件，就完成了。

5. IntelliJ IDEA

使用 Libraries\Global Libraries 和 Configure Library 对话框，可以引入 JAR 文件。

6. NetBeans

右击项目 libraries 节点，选择 Add JAR/Folder 选项，然后在浏览窗口中打开 neo4j-community\lib 文件夹并选择所有 JAR 文件。

使用依赖关系管理来添加 Neo4j JAR 包的依赖。

7. Maven

按照下面的代码段向项目添加依赖。通常是添加在项目根目录下的 pom.xml 文件中。

【程序 4-1】Neo4j 依赖代码

```
<project>
 <dependencies>
  <dependency>
   <groupId>org.neo4j</groupId>
   <artifactId>neo4j</artifactId>
   <version>3.1</version>
  </dependency>
 </dependencies>
</project>
```

对于上面代码中的 artifactId，可以从表 4-1 中找到对应项。

8. Eclipse 环境中的 Maven

对于 Eclipse 中的开发，建议安装 m2e 插件，并让 Maven 来管理项目。通过安装 m2e 插件既方便于通过 Maven 的命令行构建项目，也能够提供一个有效的 Eclipse 设置方式，便于后续的开发。

9. Ivy 配置

在 ivysettings.xml 文件中的配置代码如【程序 4-2】所示。

【程序 4-2】Ivy 依赖代码

```
<ivysettings>
  <settings defaultResolver="main"/>
  <resolvers>
    <chain name="main">
      <filesystem name="local">
        <artifact pattern="${ivy.settings.dir}/repository/[artifact]-
[revision].[ext]" />
      </filesystem>
      <ibiblio name="maven_central" root="http://repo1.maven.org/maven2/"
m2compatible="true"/>
    </chain>
  </resolvers>
</ivysettings>
```

下一步再在 ivy.xml 文件内加入如下代码。

【程序 4-3】Ivy 依赖代码

```
<dependencies>
  ...
  <dependency org="org.neo4j" name="neo4j" rev="3.1"/>
  ...
</dependencies>
```

其中 name 属性值，可从表 4-1 中找到对应项。

10. Gradle 环境配置：

用引入 Neo4j JAR 包的 gradle 构建脚本如下所示。

【程序 4-4】gradle 依赖代码

```
def neo4jVersion = "3.1"
apply plugin: 'java'
repositories {
   mavenCentral()
}
dependencies {
   compile "org.neo4j:neo4j:${neo4jVersion}"
}
```

4.2.1.2　启动和关闭 Neo4j

在将 JAR 包导入到 Java 工程中后，尝试使用 Java API 启动和关闭 Neo4j。

要创建一个新的数据库或者打开一个已经存在的数据库，首先，需要创建一个 GraphDatabaseService 实例。

```
graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( DB_PATH );
registerShutdownHook( graphDb );
```



GraphDatabaseService 实例可以在多个线程之间共享，但不能创建指向同一数据库的多个实例。

要关闭一个已经打开的实例，需要调用 shutdown()方法。

```
graphDb.shutdown();
```

为了保证 Neo4j 已经恰当地关闭，可以在代码中使用回调方法，如【程序 4-5】所示。

【程序 4-5】关闭回调方法

```
private static void registerShutdownHook( final GraphDatabaseService
graphDb )
{
    // 为 Neo4j 实例注册一个关闭操作回调方法，用来保证数据库被恰当关闭
    Runtime.getRuntime().addShutdownHook( new Thread()
    {
        @Override
        public void run()
        {
            graphDb.shutdown();
        }
    } );
}
```

1. 按照配置文件启动 Neo4j

Neo4j 数据库包含一个配置文件，如果想要让 Neo4j 按照配置启动，那么代码如下所示。

```
GraphDatabaseService graphDb = new GraphDatabaseFactory()
    .newEmbeddedDatabaseBuilder( testDirectory.graphDbDir() )
    .loadPropertiesFromFile( pathToConfig + "neo4j.conf" )
    .newGraphDatabase();
```

当然，配置文件中的配置项也完全可以在 Java 代码中设置：

```
GraphDatabaseService graphDb = new GraphDatabaseFactory()
    .newEmbeddedDatabaseBuilder( testDirectory.graphDbDir() )
    .setConfig( GraphDatabaseSettings.pagecache_memory, "512M" )
    .setConfig( GraphDatabaseSettings.string_block_size, "60" )
    .setConfig( GraphDatabaseSettings.array_block_size, "300" )
    .newGraphDatabase();
```

2. 启动一个只读实例

如果想以只读的方式打开 Neo4j 数据库而不想进行任何写操作，则代码如下所示。

```
graphDb = new GraphDatabaseFactory().newEmbeddedDatabaseBuilder( dir )
      .setConfig( GraphDatabaseSettings.read_only, "true" )
      .newGraphDatabase();
```



无论只读还是读写方式创建实例，针对一个 Neo4j 数据库 GraphDatabaseService 实例只能创建一个。

4.2.1.3 其他基于 JVM 的嵌入式开发

Neo4j 社区有一大群成员在积极贡献，他们早已参与研发了很多其他开发语言和框架的嵌入式开发模式（如：Scala、JRuby 等）。这都是通过针对特定开发语言对 Neo4j API 再封装完成的，因为他们都是基于 JVM 的，所以可以将 Neo4j 核心 API 利用 JVM 改编为自己开发语言的 API。图 4-3 展示了这些基于 JVM 的开发语言 API 的框架。

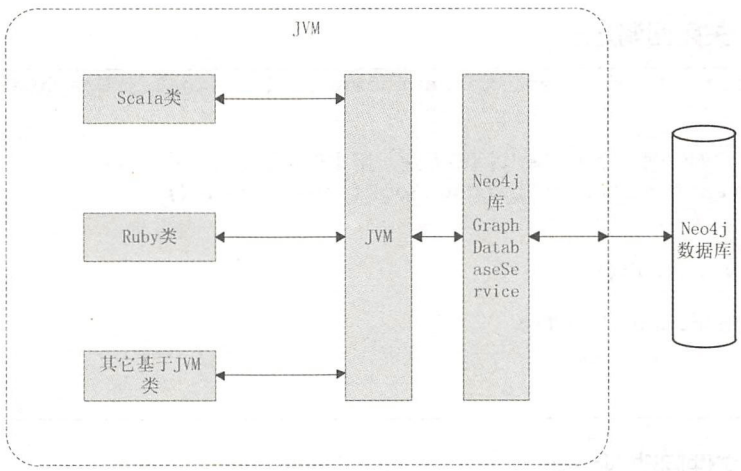


图 4-3 其他基于 JVM 语言开发结构图

需要了解的是，对于这些基于 JVM 的其他语言的封装，由于是针对 Neo4j Java API 的再封装，因此在 Java 嵌入式开发中所调用的类、方法、属性，在其他基于 JVM 的开发语言中也是可以同样使用的。

4.2.2　创建 Neo4j 图实例

上一小节中已经准备好了 Java 开发的必要工作，在这一小节中将创建并读写节点和关系。首先需要了解 Neo4j 数据库中数据的基本结构：

- 节点：表示一个实体，可以用关系连接起来。
- 关系：用来连接节点。
- 属性：依附在节点或关系上的属性及其属性值。

任何关系应当有一个指定的类型名，例如，当创建一个人际关系网络的时候人与人之间的关系类型为“认识（KNOWS）”，就是指某人认识某人，其中单个人为节点，认识为关系的类型。如果两个节点被 KNOWS 类型的关系相互连接起来，那么说明这两个人互相认识对方。图的很多含义被用这样的方式编码后存储在 Neo4j 数据库中，尽管关系是直接将两个节点连接起来的，但不管关系指向哪个节点，它们在数据库中都可以被快速地遍历、查询。

接下来的代码展示了怎样创建一个简单的图，这个图中包括了两个节点，节点之间用关系相互连接，并且节点和关系上都附有各自的属性。

4.2.2.1　准备创建一个数据库

下面我们将以 Neo4j 官方电影图数据库作为实例来介绍 Java 嵌入式开发。

在 Java 中关系的类型标签可以用枚举类型 enum 创建。在下面的例子中仅仅需要一个关系类型就可以了。

【程序 4-6】类型标签

```
private static enum RelTypes implements RelationshipType
{
ACTED_IN,
DIRECTED,
PRODUCED
}
```

接下来是启动数据库实例，需要注意的是，如果创建的数据库当前并不存在，那么系统会自动创建一个新的数据库。

```
graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( DB_PATH );
registerShutdownHook( graphDb );
```



每次启动数据库都是极其消耗系统资源的，所以不要像操作关系数据库那样每次操作都启动一个新的实例，只要创建了一个实例，则这个实例可以被多个进程共享。但要记住，在进行事务操作的时候多个进程之间是相互冲突的，在同一时间只能有一个进程操作某实例。

4.2.2.2 将操作写进事务中

在前面的介绍中，我们介绍了打开和关闭数据库的基本操作，下面将介绍事务中如何操作数据库。

在 Neo4j Java API 中所有的操作都必须放进一个事务中。正如传统关系数据库操作的原子性那样，Neo4j 也特意这样设计以保证在企业级应用中某些重要操作的可靠性，所以，在 Neo4j 中事务也是自然而然该有的，代码如下。

【程序 4-7】事务 try、catch 块

```
try ( Transaction tx = graphDb.beginTx() )
{
    // 本章所有的操作放入到这个 try、catch 块中
    tx.success();
}catch(Exception e){
    tx.failure();
}finally{
    tx.finish();
}
```

关于事务，我们将在后续的章节中给出详细的介绍，要记住本章后面所有的操作均放入到 try、catch 块中。

4.2.2.3 创建一个节点

除了上面的枚举类型定义，还需要定义变量，定义完变量后就可以对变量赋值并创建节点了。

【程序 4-8】创建节点

```
GraphDatabaseService graphDb;
Node TheMatrix;
Node Keanu;
TheMatrix = graphDb.createNode();
Keanu= graphDb.createNode();
```

创建完成后，由于并没有赋任何属性值，所以节点只包含默认的 id 值，如图 4-4 所示。

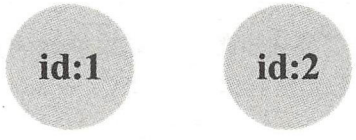


图 4-4 创建的简单节点

4.2.2.4 为节点创建属性值

在 Neo4j 中节点、关系都有属性，也就是说节点、关系所附带的值包括 ID、Label（关系为 Type）、属性三种。任何节点或关系都可以包含多个属性。

属性包括属性名和属性值两部分。属性名可以按照 Java 命名规范取名；属性值可以是单个值，也可以是数组，取值类型如表 4-2 所示。注意：NULL 在 Neo4j 中不可以赋值的，但可以通过 NULL 来判断属性是否存在。

表 4-2　属性取值类型和对应的取值范围

类型	描述	取值范围
boolean	布尔值	true/false
byte	8 位的整数	-128 到 127
short	16 位的整数	-32768 到 32767
int	32 位的整数	-2147483648 到 2147483647
long	64 位的整数	-9223372036854775808 到 9223372036854775807
float	IEEE754 标准的 32 位浮点数	
double	IEEE754 标准的 64 位浮点数	
char	16 位无符号整数代表的 Unicode 字符	u0000 到 uffff（0 到 65535）
string	Unicode 字符串	

在 Java 代码中怎样对节点或者关系上的属性赋值，如【程序 4-9】所示。

【程序 4-9】为节点创建属性并赋值

```
try{Transaction tx = graphDb.beginTx()) {
    TheMatrix = graphDb.createNode();
    TheMatrix.setProperty( " title", " The Matrix" );
    TheMatrix.setProperty( " released", 1999 );
    TheMatrix.setProperty( " tagline", " Welcome to the Real World" );
    Keanu= graphDb.createNode();
    Keanu.setProperty( "name", " Keanu Reeves" );
    Keanu.setProperty( " born", 1960 );
    tx.sucess();
}
```

4.2.2.5　为节点添加标签

在 Neo4j 中有一种将节点归类的方法，那就是标签（Label）。通过给节点添加相同或不同的标签可以将节点进行归类，这是非常有用的，比如上节创建的两个节点：一种是电影节点用来保存电影的相关信息，另一种是演员节点用来保存人物的相关信息，这样就可以方便地将节点归为两类。

在 Neo4j 2.0 版本后引入了内置节点标签的概念，帮助我们区分、归类节点。每个节点可以添加一个或多个标签，标签是一种文字描述。在 Java 嵌入式开发模式中，可以通过标签加载、查询所有节点。

要创建一个标签，可以通过使用 Java 枚举类型继承 Neo4j 的 Label 接口来创建，如下例所示。

```
public enum MyLabels implements Label{
    Movie,
    Person
}
```

有了标签的声明后，就可以使用这个枚举类型为节点添加标签了，如【程序 4-10】所示。

【程序 4-10】为节点添加标签

```
try{Transation tx = graphDb.beginTx()){
    TheMatrix.addLabel(MyLabels.Movie);
    Keanu.addLabel(MyLabels.Person);
    tx.success();
}
```

添加标签后的节点结构如图 4-5 所示。



图 4-5　创建带有标签的节点

节点有了标签后，在查询语句中可以通过标签来查询此类节点的集合。

```
try{Transation tx = graphDb.beginTx()){
ResourceIterable<Node> movies =GlobalGraphOperations.at(graphDb)
.getAllNodesWithLabel(MyLabels.Movie);
    tx.success();
}
```

通过上面两个例子可以看出，要添加标签，应该对选定的节点使用 addLabel()方法。为了能查找到所有具有给定标签的节点集合，可以使用 GlobalGraphOperatons 类中的 getAllNodesWithLabel()方法来获得。

如果在查找节点时，想通过节点属性和标签同时锁定节点集合，那么可以通过 findNodesByLabelAndProperty()方法来实现，如：

```
ResourceIterable<Node> movies = GlobalGraphOperations.at(graphDb)
    .findNodesByLabelAndProperty (MyLabels.Movie,"title","The Matrix");
```

4.2.2.6　创建关系

通过上面的操作我们已经创建了两个节点，并且两个节点都带有属性值，下面将创建两个节点之间的关系并为这个新创建的关系添加属性值。

在创建时可以指定关系的类型，关系的类型类似节点的标签，唯一不同的是关系的类型只能指定一个。

【程序 4-11】创建关系及其属性与类型

```
relationship = Keanu.createRelationshipTo(TheMatrix, RelTypes. ACTED_IN);
relationship.setProperty("roles","Neo");
```

4.2.2.7　输出图结果

下面程序的作用是把创建的图数据打印在控制台上：

【程序 4-12】打印图数据结果

```
System.out.print(Keanu.getProperty( "name" ) );
System.out.print( relationship.getProperty( "roles" ) );
System.out.print( secondNode.getProperty( "title" ) );
```

输出结果是：

```
Keanu Reeves Neo The Matrix
```

4.2.2.8　删除数据

下面的代码将会把【程序 4-11】创建的数据删除掉：

【程序 4-13】删除数据

```
//删除数据
Keanu.getSingleRelationship( RelTypes. ACTED_IN,
Direction.OUTGOING ).delete();
Keanu.delete();
TheMatrix.delete();
```



如果尝试删除一个带有关系边的节点是一定会失败的，必须先删除关系并确保没有任何关系指向这个节点时，才能够删除它。这是为了确保任何关系都有开始、结束节点的指向，没有空指向的关系。

4.2.2.9　关闭数据库

接下来是关闭数据库，之前我们已经介绍过怎样关闭一个数据库了。

```
graphDb.shutdown();
```

4.2.3　图数据遍历功能

上一节中，我们已经创建了一个由节点、关系和索引组成的图数据库，接下来将要介绍查询图数据的一种强大功能：图遍历功能。图遍历是以一种特殊的方式在图中按照节点之间的关系依次访问各个节点的过程。

Neo4j 遍历 API 采用的是一种基于回调的、惰性执行的机制，使用 Neo4j 的图遍历功能可以使用指定的方式对数据库进行遍历。在接下来的内容中我们会给出一些遍历的实例。

另外，还可以使用 Cypher 查询语言作为强大的声明式方法来对图进行查询，详见“第 3 章 Neo4j 之 Cypher”。

4.2.3.1 需要了解的概念

在接下来的内容中会用到以下几个概念来解释 Neo4j 的图形遍历功能。

（1）路径拓展（PathExpander）：定义将要对图数据库中的什么进行遍历，一般是指针对关系的指向和关系的类型进行遍历。

（2）顺序（Order）：例如深度优先或广度优先。

（3）唯一性（Uniqueness）：在遍历过程中，确保每个节点（关系，路径）只被遍历一次。

（4）评估器（Evaluator）：用来决定返回什么结果，以及是否停止或继续遍历当前位置。

（5）开始节点：启动遍历最先开始的节点。

Neo4j 遍历框架的结构图如图 4-6 所示。

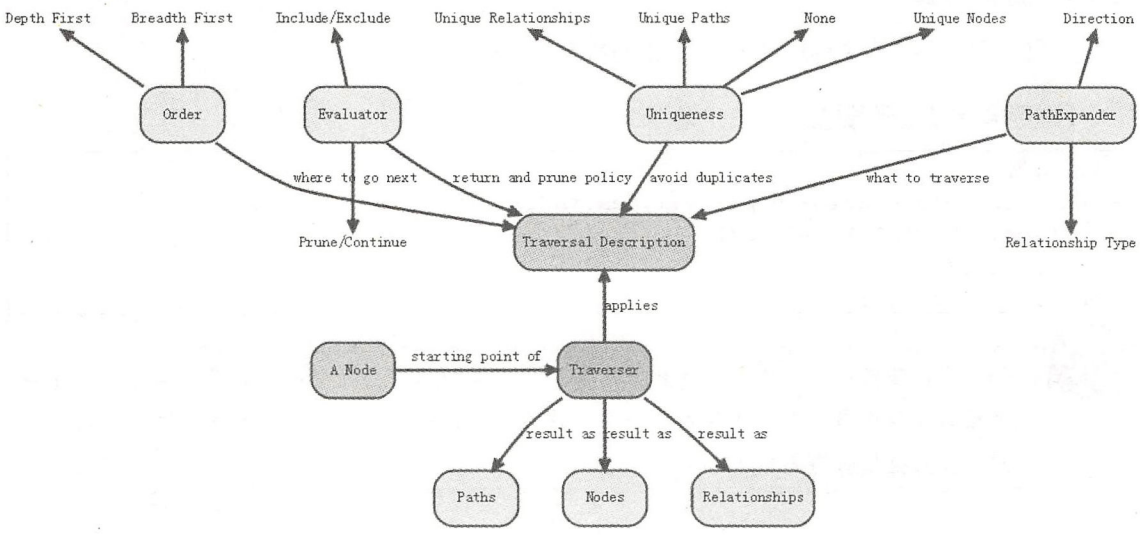


图 4-6 Neo4j 遍历框架结构图

4.2.3.2 遍历框架的 Java API

对于图 4-6 我们需要了解以下概念。

遍历框架除了包括节点和关系之外还有另外几个主要接口：遍历描述接口（Traversal Description Interface）、评估器接口（Evaluator Interface）、遍历器接口（Traverser Interface）和唯一性接口（Uniqueness Interface）。遍历框架中的路径接口（Path Interface）在遍历中有特殊用途，因为它在评估该位置时用于表示图中的位置。此外，路径拓展（替换关系拓展和普通拓展）接口是遍历的核心，但是在使用 API 时很少需要实现它。当需要对遍历顺序进行明确控制时，还有一组高级用途的接口：BranchSelector，BranchOrderingPolicy 和 TraversalBranch。

4.2.3.3 遍历描述接口

图 4-6 中的遍历描述（TraversalDescription）是用于定义和初始化遍历的主接口。它不必

由遍历框架的用户实现，而是由遍历框架的实现来作为用户描述遍历的方式。TraversalDescription 实例是不可变的，并且其方法返回一个新的 TraversalDescription，与使用该方法的参数来调用该方法的对象相比，该 TraversalDescription 可以被修改。

4.2.3.4　关系接口

图 4-6 中的关系（Relationships）接口用于将关系类型添加到要遍历的关系类型列表中。默认情况下，该列表为空，这意味着它将遍历所有类型的关系。如果将一个或多个关系添加到此列表中，则只会遍历所添加的类型的关系。有两种遍历方法，一种包括方向参数，另一种不包括方向参数，其中后者用于遍历双向关系。

4.2.3.5　评估器接口

评估器用于在每个位置（路径中的位置）处来确定：如果遍历继续，当前节点是否应包括在结果中。

给定一个路径，遍历某个分支的动作为以下动作之一：

- Evaluation.INCLUDE_AND_CONTINUE: 在结果中包括此节点并继续遍历。
- Evaluation.INCLUDE_AND_PRUNE: 在结果中包括此节点，但不继续遍历。
- Evaluation.EXCLUDE_AND_CONTINUE: 从结果中排除此节点，但继续遍历。
- Evaluation.EXCLUDE_AND_PRUNE: 从结果中排除此节点，不继续遍历。

可以为遍历器添加多个评估器。注意，对于遍历器遇到的所有位置（包括起始节点）都将调用评估器。

4.2.3.6　遍历器接口

遍历器（Traverser）对象是调用 TraversalDescription 对象的 traverse()方法返回的结果，它表示在图中定位的遍历以及结果格式的规范。每次调用 Traverser 的 next() 方法时，遍历操作将被“惰性”地执行一次。

4.2.3.7　唯一性接口

唯一性是用来设置在遍历期间如何重新访问遍历过的位置的规则。如果未设置，则默认为 NODE_GLOBAL。

可以向遍历描述提供唯一性参数以指示在什么情况下遍历可以重新访问图中的相同位置。在 Neo4j 中可使用的唯一性级别是：

（1）NONE：可以重新访问图表中的任何位置。

（2）NODE_GLOBAL 唯一性：整个图中的任何一个节点都不可能被访问多次。这可能消耗大量的内存，因为它需要保持内存中的数据结构以记住所有被访问的节点。

（3）RELATIONSHIP_GLOBAL 唯一性：整个图中的任何一个关系都不可能被访问多次。与 NODE_GLOBAL 唯一性相同，这可能会占用大量内存。由于图中的关系数量必定大于节点的数量，所以该唯一性级别的存储器开销会比 NODE_GLOBAL 更大。

（4）NODE_PATH 唯一性：节点不会先前出现在达到它的路径中。

（5）RELATIONSHIP_PATH 唯一性：先前在达到它的路径中不会存在关系边。

（6）NODE_RECENT 唯一性：类似于 NODE_GLOBAL 唯一性，存在受访节点的全局集合，每个位置被检查。然而，该唯一性级别可以指定可接受消耗存储容量的上限，大小可以通过提供一个数字作为 TraversalDescription.uniqueness()方法的第二个参数来指定，或者通过唯一性级别来指定。

（7）RELATIONSHIP_RECENT 唯一性：类似于 NODE_RECENT 唯一性，但它指的是关系而不是节点。

深度优先/广度优先就是用于设置深度优先/广度优先 BranchSelector 排序策略的遍历方法。同样的结果可以通过从 BranchOrderingPolicies 中调用带有排序策略的 order 方法来实现，也可以编写自己的 BranchSelector / BranchOrderingPolicy 来实现排序。

4.2.3.8 遍历顺序

遍历顺序就是指在遍历过程中按照什么顺序遍历图的各个分支。

这个是深度优先和广度优先方法的更通用版本，它允许将任意的分支排序策略注入描述中。

4.2.3.9 分支选择器（BranchSelector）

BranchSelector / BranchOrderingPolicy 用于选择下一次的遍历分支。这用于实现遍历排序。遍历框架提供了一些基本的排序实现：

（1）BranchOrderingPolicies.PREORDER_DEPTH_FIRST：深度优先遍历，在访问其子节点之前访问每个节点。

（2）BranchOrderingPolicies.POSTORDER_DEPTH_FIRST：深度优先遍历，访问其子节点后访问每个节点。

（3）BranchOrderingPolicies.PREORDER_BREADTH_FIRST：广度优先遍历，在访问其子节点之前访问每个节点。

（4）BranchOrderingPolicies.POSTORDER_BREADTH_FIRST：广度优先遍历，访问其子节点后访问每个节点。



广度优先遍历比深度优先遍历需要更高的内存开销。

BranchSelectors 具有状态属性，因此需要为每次遍历唯一地实例化这个类。因此，它通过 BranchOrderingPolicy 接口提供给 TraversalDescription，BranchOrderingPolicy 接口是 BranchSelector 实例的工厂。

遍历框架的用户很少需要实现自己的 BranchSelector 或 BranchOrderingPolicy，以让图算法实现者提供它们自己的遍历顺序。Neo4j 图算法包含如用于 BestFirst 搜索算法（例如 A *和 Dijkstra）中的 BestFirst 顺序 BranchSelector / BranchOrderingPolicy。



4.2.3.10　分支遍历策略

一个用于创建 BranchSelectors 的工厂，用于决定返回分支的顺序（其中分支的位置表示为从起始节点到当前节点的路径）。常见的策略是深度优先和广度优先，这就是为什么有更方便的方法。例如，调用 TraversalDescription # depthFirst()等效于：

```
description.order( BranchOrderingPolicies.PREORDER_DEPTH_FIRST );
```

1. 分支选择器（BranchSelector）

BranchSelector 用来从某个分支获取更多分支。本质上，这些是路径（Path）和关系拓展器（RelationshipExpander）的复合，可以用来从当前的一个分支获得新的 TraversalBranches。

2. 遍历路径

Path 是一个通用接口，它是 Neo4j API 的一部分。在 Neo4j 的遍历 API 中，使用 Paths 可以进行双向的遍历。遍历器可以将图中被标记为返回的、被访问位置的、路径的形式返回其结果。Path 对象也用来评估图中的位置，用于确定遍历是否应当从某个点继续，以及是否应当将某个位置包括在结果集中。

3. 路径拓展器

遍历框架使用路径拓展器（PathExpander，路径拓展器用来替换关系拓展器）来发现在遍历中从特定路径到进一步分支应遵循的关系。

4. 拓展器

这个是比注入 RelationshipExpander 关系更通用的方式，可以定义要为任何给定节点遍历的所有关系。

Expander 接口是 RelationshipExpander 接口的扩展，可以构建 Expander 的自定义版本。TraversalDescription 的实现使用它来提供用于定义要遍历的关系类型的方法，通过在 TraversalDescription 中内部构建它是使用 API 的用户定义 RelationshipExpander 的通常方式 。

所有由 Neo4j 遍历框架提供的 RelationshipExpanders 也实现了 Expander 接口。对于熟悉遍历 API 的开发者，更容易实现 RelationshipExpander 接口，因为它只包含一个方法：从路径、节点获取关系的方法，Expander 接口添加的方法仅用于构建新的拓展器。

4.2.3.11　Java 中使用遍历框架

使用遍历描述可以生成遍历器。如图 4-7 所示给出遍历实例。



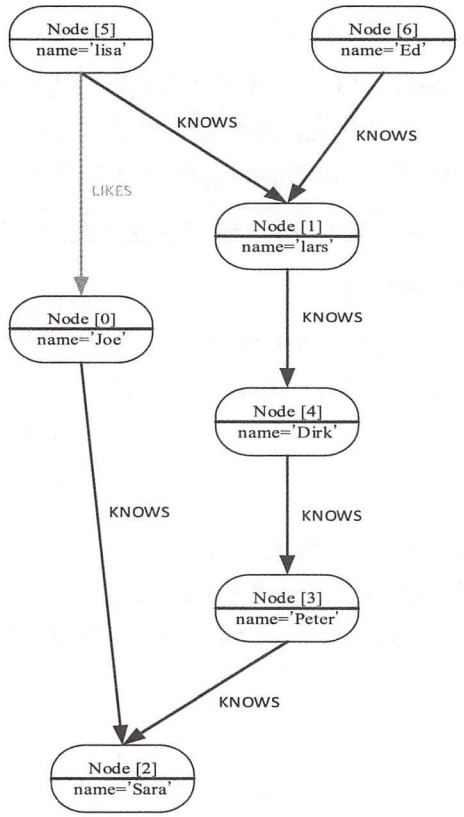


图 4-7　Neo4j 遍历实例

首先需要定义一个关系类型（RelationshipTypes）。

```
private enum Rels implements RelationshipType
{
    LIKES, KNOWS
}
```

图 4-7 可以用下面的遍历器遍历，从“Joe”节点开始。

【程序 4-14】遍历实例代码

```
for ( Path position : db.traversalDescription()
        .depthFirst()
        .relationships( Rels.KNOWS )
        .relationships( Rels.LIKES, Direction.INCOMING )
        .evaluator( Evaluators.toDepth( 5 ) )
        .traverse( node ) )
{
    output += position + "\n";
}
```

上面代码运行后，将在控制台打印如下结果。

```
(0)
(0)
(0)<--[LIKES,1]--(5)
(0)<--[LIKES,1]--(5)--[KNOWS,6]-->(1)
(0)<--[LIKES,1]--(5)--[KNOWS,6]-->(1)<--[KNOWS,5]--(6)
(0)<--[LIKES,1]--(5)--[KNOWS,6]-->(1)--[KNOWS,4]-->(4)
(0)<--[LIKES,1]--(5)--[KNOWS,6]-->(1)--[KNOWS,4]-->(4)--[KNOWS,3]-->(3)
(0)<--[LIKES,1]--(5)--[KNOWS,6]-->(1)--[KNOWS,4]-->(4)--[KNOWS,3]-->(3)--
[KNOWS,2]-->(2)
```

由于遍历的描述是不可变的，因此可以创建一个模板描述来保存由不同遍历共享的公共设置，例如：

```
friendsTraversal = db.traversalDescription()
       .depthFirst()
       .relationships( Rels.KNOWS )
       .uniqueness( Uniqueness.RELATIONSHIP_GLOBAL );
```

这个遍历器将产生以下输出（我们将继续从“Joe”节点开始）。

```
(0)
(0)--[KNOWS,0]-->(2)
(0)--[KNOWS,0]-->(2)<--[KNOWS,2]--(3)
(0)--[KNOWS,0]-->(2)<--[KNOWS,2]--(3)<--[KNOWS,3]--(4)
(0)--[KNOWS,0]-->(2)<--[KNOWS,2]--(3)<--[KNOWS,3]--(4)<--[KNOWS,4]--(1)
(0)--[KNOWS,0]-->(2)<--[KNOWS,2]--(3)<--[KNOWS,3]--(4)<--[KNOWS,4]--(1)<--
[KNOWS,6]--(5)
(0)--[KNOWS,0]-->(2)<--[KNOWS,2]--(3)<--[KNOWS,3]--(4)<--[KNOWS,4]--(1)<--
[KNOWS,5]--(6)
```

现在让我们创建一个新的遍历器，将遍历深度限制为 3。

```
for ( Path path : friendsTraversal
       .evaluator( Evaluators.toDepth( 3 ) )
       .traverse( node ) )
{
   output += path + "\n";
}
```

上面程序运行结果为：

```
(0)
(0)--[KNOWS,0]-->(2)
(0)--[KNOWS,0]-->(2)<--[KNOWS,2]--(3)
(0)--[KNOWS,0]-->(2)<--[KNOWS,2]--(3)<--[KNOWS,3]--(4)
```

尝试将遍历深度设为 2 到 4：

```
for ( Path path : friendsTraversal
       .evaluator( Evaluators.fromDepth( 2 ) )
       .evaluator( Evaluators.toDepth( 4 ) )
       .traverse( node ) )
{
```



```
    output += path + "\n";
}
```

结果为：

```
(0)--[KNOWS,0]-->(2)<--[KNOWS,2]--(3)
(0)--[KNOWS,0]-->(2)<--[KNOWS,2]--(3)<--[KNOWS,3]--(4)
(0)--[KNOWS,0]-->(2)<--[KNOWS,2]--(3)<--[KNOWS,3]--(4)<--[KNOWS,4]--(1)
```

可以将遍历器转换为可迭代的节点，如下所示。

```
for ( Node currentNode : friendsTraversal
        .traverse( node )
        .nodes() )
{
    output += currentNode.getProperty( "name" ) + "\n";
}
```

在上面这个例子中，遍历的名字结果为：

```
Joe
Sara
Peter
Dirk
Lars
Lisa
Ed
```

使用关系来遍历也是可以的，如下面例子：

```
for ( Relationship relationship : friendsTraversal
        .traverse( node )
        .relationships() )
{
    output += relationship.getType().name() + "\n";
}
```

上面运行结果为：

```
KNOWS
KNOWS
KNOWS
KNOWS
KNOWS
KNOWS
```

4.2.4 数据索引

Neo4j 本身可通过有效的方法快速遍历图，如果能够提供一种索引帮助 Neo4j 在进行遍历的时候提高查询效率那就再好不过了。

在关系型数据库中，索引提供了有序排列数据的方式，使用索引可以像字典一样快速定位所要查找的记录。在 Neo4j 中索引也可以对指定的属性值进行快速定位查找，与关系数据



仅供非商业用途或交流学习使用

库所不同的是 Neo4j 中除了 Cypher，也可以使用 Java 应用程序通过代码来创建索引。

在 Neo4j 中索引分为自动索引和手动索引两种，对于自动索引就是使用在“3.5.1 索引”小节中介绍的使用 Cypher 语句创建的索引；对于手动索引就是接下来我们将要介绍的使用 Java 应用程序通过代码来创建索引。在接下来的内容中，将介绍如何使用 Java API 对 Neo4j 数据库进行创建、维护和使用索引。

4.2.4.1　自动索引

请参见“3.5.1 索引”小节，本节不再对自动索引进行介绍。

4.2.4.2　手动索引

从 Neo4j 2.0 开始，手动索引不再是 Neo4j 索引数据的首选方法，建议在数据库模式中定义索引，也就是使用 Cypher 创建索引。但是，Neo4j 仍然支持手动索引，因为目前某些功能（如全文搜索）尚不能由 Cypher 索引处理。

手动索引操作是 Neo4j 索引 API 的一部分。每个索引都绑定到某个唯一的属性名称上（例如“first_name”或“books”），并且对节点或关系都可以创建索引。

默认索引实现由 neo4j-lucene-index 组件提供，这个组件可以在 Neo4j 官方网站下载，也可以从 Maven 网站[11]下载。对于 Maven 用户，neo4j-lucene-index 组件的搜索标签为：org.neo4j: neo4j-lucene-index，在创建 Maven 工程时通过这个标签我们可以用 Maven 自动添加 JAR 包。并且应该使用与我们的 Neo4j 相同版本的 org.neo4j:neo4j-kernel 包。不同版本的索引和内核组件在一般情况下是不兼容的。这两个组件在 Maven 中都可以通过标签：org.neo4j:neo4j:pom 获得，并且是版本兼容的，这样就能很容易地保持版本的同步了。

与 Neo4j 中的任何修改操作一样，所有修改索引的操作必须在事务内部执行。

4.2.4.3　创建索引

如果某个索引在请求使用时发现它并不存在，则系统会自动创建此索引。如果没有给它定义配置参数，那么索引将使用默认配置参数创建。

下面让我们创建一些索引。

【程序 4-15】创建索引

```
IndexManager index = graphDb.index();
Index<Node> actors = index.forNodes( "actors" );
Index<Node> movies = index.forNodes( "movies" );
RelationshipIndex roles = index.forRelationships( "roles" );
```

上面代码将创建两个节点索引和一个关系索引，也就是为演员和电影类节点创建的索引，另一个是为角色关系创建的索引，并且这些索引都是使用默认配置创建的。

如果想要知道某个索引是否已经存在，可以通过如下代码实现。

```
IndexManager index = graphDb.index();
boolean indexExists = index.existsForNodes( "actors" );
```

[11] http://repo1.maven.org/maven2/org/neo4j/neo4j-lucene-index/



仅供非商业用途或交流学习使用

4.2.4.4 删除索引

索引是可以删除的。删除时，将删掉索引的全部内容及其关联配置。删除这个索引后，可以再使用相同的名称创建其他索引，这样能确保索引的唯一性。

删除索引的代码如下：

【程序 4-16】删除索引

```
IndexManager index = graphDb.index();
Index<Node> actors = index.forNodes( "actors" );
actors.delete();
```



索引的实际删除是在事务内提交的。在 delete()方法被调用后再对这个索引实例的调用无论是在该事务内部还是外部都是无效的，但是如果这个事务被回滚了，则对索引的删除操作就无效了，索引还会依然存在。

4.2.4.5 添加索引

索引支持将任意数量的键值对与任意数量的实体（节点或关系）相关联，也就是说一个创建好的索引可以被赋予在任何数量的节点或关系上。

我们使用在上一节中创建的索引将其添加在几个节点上：

【程序 4-17】将索引添加到节点

```
// 演员节点
Node reeves = graphDb.createNode();
reeves.setProperty( "name", "Keanu Reeves" ); //关联字符串值
actors.add( reeves, "name", reeves.getProperty( "name" ) ); //关联节点
Node bellucci = graphDb.createNode();
bellucci.setProperty( "name", "Monica Bellucci" );
actors.add( bellucci, "name", bellucci.getProperty( "name" ) );
// 在本例中我们仅用作搜索，所以 name 索引被关联多个值，包括节点和字符串值.
actors.add( bellucci, "name", "La Bellucci" );
// 电影节点
Node theMatrix = graphDb.createNode();
theMatrix.setProperty( "title", "The Matrix" );
theMatrix.setProperty( "year", 1999 );
movies.add( theMatrix, "title", theMatrix.getProperty( "title" ) );
movies.add( theMatrix, "year", theMatrix.getProperty( "year" ) );
Node theMatrixReloaded = graphDb.createNode();
theMatrixReloaded.setProperty( "title", "The Matrix Reloaded" );
theMatrixReloaded.setProperty( "year", 2003 );
movies.add(                    theMatrixReloaded,                    "title",
theMatrixReloaded.getProperty( "title" ) );
movies.add( theMatrixReloaded, "year", 2003 );
Node malena = graphDb.createNode();
```

仅供非商业用途或交流学习使用


```
malena.setProperty( "title", "Malèna" );
malena.setProperty( "year", 2000 );
movies.add( malena, "title", malena.getProperty( "title" ) );
movies.add( malena, "year", malena.getProperty( "year" ) );
```

上面例子是将索引关联到节点上的，下面是将索引关联到关系上的：

【程序 4-18】将索引添加到关系

```
// 我们需要关系上有类型值
RelationshipType ACTS_IN = RelationshipType.withName( "ACTS_IN" );
// 创建关系
Relationship role1 = reeves.createRelationshipTo( theMatrix, ACTS_IN );
role1.setProperty( "name", "Neo" );
roles.add( role1, "name", role1.getProperty( "name" ) );
Relationship role2 = reeves.createRelationshipTo( theMatrixReloaded,
ACTS_IN );
role2.setProperty( "name", "Neo" );
roles.add( role2, "name", role2.getProperty( "name" ) ); //关联关系
Relationship role3 = bellucci.createRelationshipTo( theMatrixReloaded,
ACTS_IN );
role3.setProperty( "name", "Persephone" );
roles.add( role3, "name", role3.getProperty( "name" ) ); //关联关系
Relationship role4 = bellucci.createRelationshipTo( malena, ACTS_IN );
role4.setProperty( "name", "Malèna Scordia" );
roles.add( role4, "name", role4.getProperty( "name" ) ); //关联关系
```

通过创建上面的索引后，数据库中的图结构如图 4-8 所示。

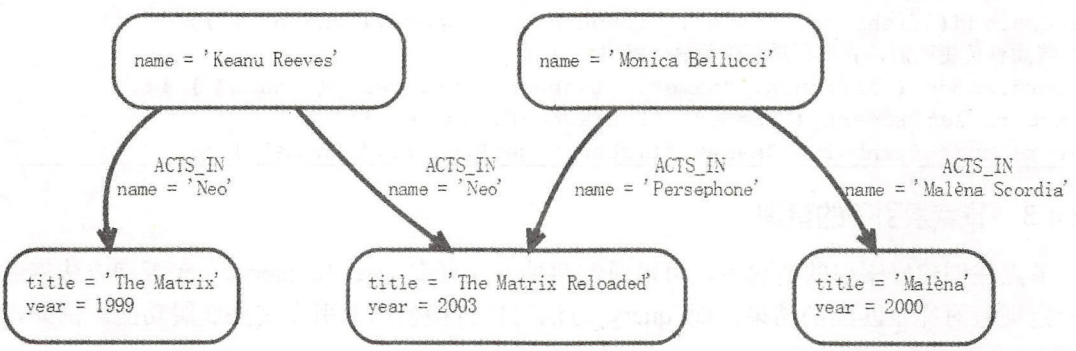


图 4-8　创建索引后的图结构

4.2.4.6　将索引移除

将索引从节点或关系上移除与上面的添加操作类似，移除索引需要指定索引的参数，可以通过提供以下参数组合来完成：

- 实体（节点、索引）
- 实体、索引键名



仅供非商业用途或交流学习使用

- 实体、索引键名、索引键值

【程序 4-19】移除索引

```
// 从bellucci上完全移除演员索引（actors index）
actors.remove( bellucci );
// 从任何以"name"为索引键的节点或关系上移除演员索引（actors index）
actors.remove( bellucci, "name" );
// 从索引键为"name"索引值为"La Bellucci"的节点或关系上移除演员索引
actors.remove( bellucci, "name", "La Bellucci" );
```

4.2.4.7 更新索引

要更新索引条目，必须删除旧的索引条目，然后再添加新的索引条目。（有关删除索引条目的详细信息请参见“4.2.4.6 将索引移除”小节）。

请记住，节点或关系可以与索引中任意数量的键值对关联。这意味着可以使用具有相同键的许多键值对对节点或关系建立索引。当属性值改变想更新索引时，仅仅对新值创建索引是不够的，还必须删除旧值。

下面的代码展示如何更新索引：

【程序 4-20】更新索引

```
// 创建一个带有属性的节点
// 然后我们再去更新这个属性
Node fishburn = graphDb.createNode();
fishburn.setProperty( "name", "Fishburn" );
//对 name 创建索引
actors.add( fishburn, "name", fishburn.getProperty( "name" ) );
// 当属性值更改后，我们再重新更新这个索引
actors.remove( fishburn, "name", fishburn.getProperty( "name" ) );
fishburn.setProperty( "name", "Laurence Fishburn" );
actors.add( fishburn, "name", fishburn.getProperty( "name" ) );
```

4.2.4.8 节点索引下的查询

节点在创建好索引的情况下，可以通过两种方式查询：get 和 query。get 查询方法将返回与给定键值对完全匹配的结果，而 query 方法可以直接使用索引查询更低层功能。例如，使用 Lucene 模糊匹配查询语法。

1. get 方法

下面是一个使用 get 方法查询的例子，它将返回与查询的键值（"name", "Keanu Reeves"）对完全匹配的结果。

```
IndexHits<Node> hits = actors.get( "name", "Keanu Reeves" );
Node reeves = hits.getSingle();
```

IndexHits 是一个 Iterable 类的继承，它提供一些特别有用的方法。例如，getSingle()返回结果迭代器中的第一个，也是唯一的项，如果没有任何相匹配的数据，则返回 null。



仅供非商业用途或交流学习使用

get()方法返回的是个 list 类型的结果集，以下是如何通过精确匹配获取单个关系并检索其开始和结束节点：

```
Relationship persephone = roles.get( "name", "Persephone" ).getSingle();
Node actor = persephone.getStartNode();
Node movie = persephone.getEndNode();
```

也可以用 for 循环迭代输出所有结果集：

```
for ( Relationship role : roles.get( "name", "Neo" ) )
{
    Node reeves = role.getStartNode();
}
```

2. query 方法

query 方法有两种使用方式，一种方式是提供单一的键值对来匹配查询索引所关联的属性；另一种方式是提供多个键值对来匹配。在值中我们可以使用模糊匹配运算符“*”。

使用单一的键值对来匹配查询索引所关联的属性，代码如下：

```
for ( Node actor : actors.query( "name", "*e*" ) )
{
    // *e*为模糊匹配查询，返回结果为：Reeves 和 Bellucci
}
```

使用多个键值对来匹配查询索引所关联的属性，代码如下：

```
for ( Node movie : movies.query( "title:*Matrix* AND year:1999" ) )
{
    // 返回结果为 1999年的 "The Matrix".
}
```



查询键值对中取值含有*或? 属于模糊匹配查询，具体用法参见“第 3 章 Neo4j 之 Cypher”。

4.2.4.9　关系索引下的查询

关系索引下的查询与节点的索引类似，但在查询中需要指定所要查询的关系的开始节点或结束节点。这些额外的方法驻留在 RelationshipIndex 接口中，该接口扩展了 Index <Relationship>接口。

下面是关系索引下查询的示例，本例中使用上一节中 reeves 作为开始节点。

【程序 4-21】关系索引示例

```
// 查询一个以 reeves 作为开始节点的关系
// 使用单一键值对的匹配方法
IndexHits<Relationship> reevesAsNeoHits;
reevesAsNeoHits = roles.get( "name", "Neo", reeves, null );
Relationship reevesAsNeo = reevesAsNeoHits.iterator().next();
```



仅供非商业用途或交流学习使用


```
reevesAsNeoHits.close();
//查询一个以 theMatrix 作为结束节点的关系
// 使用 query 方法
IndexHits<Relationship> matrixNeoHits;
matrixNeoHits = roles.query( "name", "*eo", null, theMatrix );
Relationship matrixNeo = matrixNeoHits.iterator().next();
matrixNeoHits.close();
```

以下是查询特定关系类型的示例：

【程序 4-22】查询特定关系类型示例

```
//查询一个以指定节点作为结束节点的关系
//本例使用了关系的类型 type.
//下面 add 方法是将关系添加到索引:
roles.add( reevesAsNeo, "type", reevesAsNeo.getType().name() );
// 我们需要先提交上面添加索引的操作，使其在数据库中生效然后才能做查询
tx.success();
tx.close();
// 下面是查询代码:
try ( Transaction tx = graphDb.beginTx() )
{
   IndexHits<Relationship> typeHits = roles.query( "type:ACTS_IN AND
name:Neo", null, theMatrix );
   Relationship typeNeo = typeHits.iterator().next();
typeHits.close();
}
```

如果在数据库中节点之间具有非常多的关系，那么在这种情况下上面的查询是非常有效的，因为它减少了两个节点间关系的查询时间。

4.2.4.10 结果评分

在查询中特别是模糊匹配查询中，我们需要得到结果集中每个结果的匹配相似度，这就需要评分功能（Scores）实现。具体请看下面例子：

```
IndexHits<Node> hits = movies.query( "title", "The*" );
for ( Node movie : hits )
{
   System.out.println( movie.getProperty( "title" ) + " " +
hits.currentScore() );
}
```

4.2.4.11 索引配置和全文索引

在创建索引时，可以配置索引的一些属性来控制索引的行为。例如创建一个 Lucene 全文索引。

【程序 4-23】全文索引示例

```
IndexManager index = graphDb.index();
```



仅供非商业用途或交流学习使用


```
Index<Node> fulltextMovies = index.forNodes( "movies-fulltext",
      MapUtil.stringMap( IndexManager.PROVIDER, "lucene", "type",
"fulltext" ) );
fulltextMovies.add( theMatrix, "title", "The Matrix" );
fulltextMovies.add( theMatrixReloaded, "title", "The Matrix Reloaded" );
// 在全文索引中搜索
Node found = fulltextMovies.query( "title", "reloAdEd" ).getSingle();
```

下面例子是创建一个不区分大小写的全文索引例子。

【程序 4-24】不区分大小写的全文索引

```
Index<Node> index = graphDb.index().forNodes( "exact-case-insensitive",
      MapUtil.stringMap( "type", "exact", "to_lower_case", "true" ) );
Node node = graphDb.createNode();
index.add( node, "name", "Thomas Anderson" );
assertContains( index.query( "name", "\"Thomas Anderson\"" ), node );
assertContains( index.query( "name", "\"thoMas ANDerson\"" ), node );
```

在上面代码中为了实现模糊匹配我们使用了 query 查询方法，如果要实现对 name 属性值的完全匹配查询则需要使用 get 查询方法。

4.2.4.12　Lucene 索引的其他特性

1. 数值范围

当对数值型数据创建索引时，Lucene 支持针对数值的智能索引。首先需要使用 ValueContext 方法标记一个值，使其被当作一个数值来创建索引，下面代码针对年份数值创建了 Luncene 索引，然后按照年份范围进行查询，如下所示。

【程序 4-25】Lucene 数值索引

```
movies.add( theMatrix, "year-numeric", new
ValueContext( 1999 ).indexNumeric() );
movies.add( theMatrixReloaded, "year-numeric", new
ValueContext( 2003 ).indexNumeric() );
movies.add( malena, "year-numeric", new
ValueContext( 2000 ).indexNumeric() );
int from = 1997;
int to = 1999;
hits = movies.query( QueryContext.numericRange( "year-numeric", from,
to ) );
```

2. Lucene 排序

Lucene 索引具有优秀的排序功能，通过 QueryContext 类就可以实现，具体代码如下所示。

【程序 4-26】Lucene 排序

```
hits = movies.query( "title", new QueryContext( "*" ).sort( "title" ) );
for ( Node hit : hits )
```



仅供非商业用途或交流学习使用


```
{
    // 按照 title 排序
}
hits = movies.query( new QueryContext( "title:*" ).sort( "year",
"title" ) );
for ( Node hit : hits )
{
    //首先按照 year 排序然后再按照 title
}
```

也可以按照匹配的相似度（分数）对结果进行排序：

```
hits = movies.query( "title", new QueryContext( "The*" ).sortByScore() );
for ( Node movie : hits )
{
    // 按照相似度排序
}
```

3. 使用 Lucene 查询对象进行查询

可以通过编程方式实例化这些查询并作为参数传入，而不是传递 Lucene 查询语法查询。例如：

```
Node actor = actors.query( new TermQuery( new Term( "name", "Keanu
Reeves" ) ) ).getSingle();
```



TermQuery 基本上与在索引上使用 get 方法是一样的。

下面是如何使用 Lucene 查询对象执行通配符搜索：

```
hits = movies.query( new WildcardQuery( new Term( "title", "The
Matrix*" ) ) );
for ( Node movie : hits )
{
    System.out.println( movie.getProperty( "title" ) );
}
```

4. 复合查询

Lucene 支持在同一查询中查询多个术语，如下所示。

```
hits = movies.query( "title:*Matrix* AND year:1999" );
```

复合查询无法对已经创建索引条目和尚未创建索引条目的属性同时进行搜索。

5. 操作符

查询中的默认关系运算符（即：AND 或 OR） 也可以通过 QueryContext 类来更改该行为：

```
QueryContext query = new QueryContext( "title:*Matrix* year:1999" )
        .defaultOperator( Operator.AND );
hits = movies.query( query );
```



4.2.5　过程

过程（Procedure）是以 Java 编写然后部署到数据库中的扩展插件，过程可以在 Cypher 中调用。我们可以将用户自定义过程与关系数据库的存储过程相联系起来，但 Neo4j 的用户自定义过程与关系数据库的存储过程又有本质的区别：Neo4j 的用户自定义过程并不是用 Cypher 创建的数据操作的集合，而是使用 Java API 创建的数据库功能插件。

过程是一种允许 Neo4j 通过编写自定义代码来扩展出更多功能的机制，然后直接通过 Cypher 调用。过程可以接收参数对数据库执行操作并返回相应结果。

过程用 Java 编写并编译成 JAR 文件。可以通过将 JAR 文件放入每个独立或集群服务器上的$NEO4J_HOME/plugins 目录中来将它们部署到 Neo4j 数据库中。部署完后每个服务器必须重新启动数据库以使新的过程生效。

过程是扩展 Neo4j 的首选手段。过程能够提供的功能如下：

- 提供对 Cypher 中不可用的功能的访问，例如手动索引。
- 提供对第三方系统的访问。
- 执行全局操作，例如对连接的组件计数或查找密集节点。
- 实现难以用 Cypher 明确表达的操作。

4.2.5.1　调用过程

调用过程，需要使用 Cypher CALL 子句。过程名称必须是唯一指定的，如使用以下方法调用在包 org.neo4j.examples 中定义的名为 findDenseNodes 的过程：

```
CALL org.neo4j.examples.findDenseNodes(1000)
```

CALL 语句可以是 Cypher 语句中的唯一子句，或者可以与其他子句组合。可以在查询中直接提供参数或从关联的参数集中提取参数。有关调用过程的更多详细信息，请参见“3.3.21 CALL”小节。

4.2.5.2　Neo4j 内建的过程

Neo4j 本身带有一些内建的过程。表 4-3 给出常用的一些内建过程，其他更多的内建过程介绍请参见“附录 B Neo4j 内建过程”。

表 4-3　Neo4j 内建过程

过程名字	调用命令	功能
ListLabels	CALL db.labels()	列出数据库中存在的所有 label
CALL db.labels()	CALL db.relationshipTypes()	列出数据库中存在的所有关系的 type
ListPropertyKeys	CALL db.propertyKeys()	列出数据库中存在的所有属性名
ListIndexes	CALL db.indexes()	列出数据库中存在的所有索引
ListConstraints	CALL db.constraints()	列出数据库中存在的所有约束
ListProcedures	CALL dbms.procedures()	列出数据库中存在的所有过程
ListComponents	CALL dbms.components()	列出数据库中存在的所有 DBMS 组件及其版本号
QueryJmx	CALL dbms.queryJmx(query)	用域名和名称查询 JMX 管理信息，例如“org.neo4j.*”
AlterUserPassword	CALL dbms.changePassword(query)	修改账户密码



4.2.5.3 用户自定义过程

下面讨论的示例可以从 GitHub 开源社区[12]上下载。如果有 GitHub 账户，可以单击 fork 按钮存储到自己的账户下并使用其代码。

用户自定义过程是用 Java 编程语言编写的。自定义过程通过包含代码本身以及任何依赖包（不包括 Neo4j）的 JAR 文件进行部署。这些文件应放置在每个独立数据库或集群成员的 plugin 目录中，并在下次重新启动数据库后生效。

下面的示例将介绍创建和部署新过程的步骤。

1. 创建一个新 Java 项目

首先编译编写好的代码，然后生成一个 JAR 文件。下面是使用 Maven 构建项目的示例配置。为了可读性，这里仅介绍 Maven pom.xml 文件的部分内容，也可以从 GitHub 上将这个文件模板下载下来。

【程序 4-27】pom.xml 文件

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>org.neo4j.example</groupId>
<artifactId>procedure-template</artifactId>
<version>1.0.0-SNAPSHOT</version>
<packaging>jar</packaging>
<name>Neo4j Procedure Template</name>
<description>A template project for building a Neo4j Procedure</description>
<properties>
<neo4j.version>3.1</neo4j.version>
</properties>
```

接下来，定义构建依赖关系的 XML。相关配置都必须在 pom.xml 中<dependencies> </dependencies>标签之间。

第一个依赖项部分包括过程在运行时使用的过程 API。范围设置“scope”为“provided”，因为一旦这个过程部署到 Neo4j 实例中，此依赖将由 Neo4j 提供。

```
<dependency>
<groupId>org.neo4j</groupId>
<artifactId>neo4j</artifactId>
<version>${neo4j.version}</version>
<scope>provided</scope>
</dependency>
```

接下来，添加测试所需的依赖库：

（1）Neo4j Harness：一个允许启动轻量级 Neo4j 实例的实用程序。它用于启动 Neo4j，

[12] https://github.com/neo4j-examples/neo4j-procedure-template

其中部署了特定的过程。

（2）Neo4j Java 驱动程序：用于发送调用过程的 Cypher 语句。

（3）JUnit，一个通用的Java测试框架。

```
<dependency>
<groupId>org.neo4j.test</groupId>
<artifactId>neo4j-harness</artifactId>
<version>${neo4j.version}</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.neo4j.driver</groupId>
<artifactId>neo4j-java-driver</artifactId>
<version>1.0-SNAPSHOT</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.12</version>
<scope>test</scope>
</dependency>
```

如果使用 Gradle 添加依赖包，则配置如下：

【程序 4-28】Gradle 配置文件

```
project.ext {
    neo4j_version = ""
}
dependencies {
compile group: "org.neo4j", name:"neo4j", version:project.neo4j_version
testCompile group: "org.neo4j", name:"neo4j-kernel",
version:project.neo4j_version, classifier:"tests"
testCompile group: "org.neo4j", name:"neo4j-io",
version:project.neo4j_version, classifier:"tests"
testCompile group: "junit", name:"junit", version:4.12
}
```

2. 编写集成测试

本测试实例需要添加 Neo4j Harness 和 JUnit 依赖包，这两个包可用于编写过程的集成测试和单元测试。

具体步骤是：首先，决定程序应该做什么；然后，写一个测试证明它是正确的；最后，写一个能够通过测试的过程。

下面是一个用于测试从 Cypher 访问 Neo4j 的全文索引的过程的模板。

【程序 4-29】从 Cypher 访问 Neo4j

```
package example;
```



```
import org.junit.Rule;
import org.junit.Test;
import org.neo4j.driver.v1.*;
import org.neo4j.graphdb.factory.GraphDatabaseSettings;
import org.neo4j.harness.junit.Neo4jRule;
import static org.hamcrest.core.IsEqual.equalTo;
import static org.junit.Assert.assertThat;
import static org.neo4j.driver.v1.Values.parameters;
public class ManualFullTextIndexTest
{
// 启动一个 Neo4j 实例
@Rule
public Neo4jRule neo4j = new Neo4jRule()
// 下面是我们将要测试的过程
.withProcedure( FullTextIndex.class );
@Test
public void shouldAllowIndexingAndFindingANode() throws Throwable
{
// 在下面 try 程序块中，保证在测试程序最后关闭驱动
try( Driver driver = GraphDatabase.driver( neo4j.boltURI() ,
Config.build().withEncryptionLevel(
Config.EncryptionLevel.NONE ).toConfig() ) )
{
Session session = driver.session();
// 这里先创建一个节点
long nodeId = session.run( "CREATE (p:User {name:'Brookreson'}) RETURN
id(p)" )
.single()
.get( 0 ).asLong();
//当我用 index 过程对节点创建索引
session.run( "CALL example.index({id}, ['name'])", parameters( "id",
nodeId ) );
//我就可以用 lucene 语法查询我想要找的节点
StatementResult result = session.run( "CALL example.search('User',
'name:Brook*')" );
assertThat( result.single().get( "nodeId" ).asLong(), equalTo( nodeId ) );
}
}
}
```

3. 写一个过程

通过测试，我们编写了一个满足测试期望的过程程序。完整的示例可以从 GitHub 上找到[13]。



（1）所有过程都需要使用@Procedure 注解。有写入数据库操作的过程，需要另外添加@PerformsWrites。（2）过程的上下文对象要与过程使用的每个资源对象相同，都需要添加@Context 注解。（3）需要了解过程有关输入和输出的详细信息，请参阅 API 文档[14]。

[13] https://github.com/neo4j-examples/neo4j-proceduretemplate
[14] http://neo4j.com/docs/java- reference/3.1/javadocs/index.html?org/neo4j/procedure/Procedure.html

【程序 4-30】用户自定义过程模板代码

```
package example;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.stream.Stream;
import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.Label;
import org.neo4j.graphdb.Node;
import org.neo4j.graphdb.index.Index;
import org.neo4j.graphdb.index.IndexManager;
import org.neo4j.logging.Log;
import org.neo4j.procedure.Context;
import org.neo4j.procedure.Name;
import org.neo4j.procedure.PerformsWrites;
import org.neo4j.procedure.Procedure;
import static org.neo4j.helpers.collection.MapUtil.stringMap;
/**
* This is an example showing how you could expose Neo4j's full text indexes
as
* two procedures - one for updating indexes, and one for querying by label
and
* the lucene query language.
*/
public class FullTextIndex
{
// Only static fields and @Context-annotated fields are allowed in
// Procedure classes. This static field is the configuration we use
// to create full-text indexes.
private static final Map<String,String> FULL_TEXT =
stringMap( IndexManager.PROVIDER, "lucene", "type", "fulltext" );
// This field declares that we need a GraphDatabaseService
// as context when any procedure in this class is invoked
@Context
public GraphDatabaseService db;
// This gives us a log instance that outputs messages to the
// standard log, `neo4j.log`
@Context
public Log log;
/**
* This declares the first of two procedures in this class - a
* procedure that performs queries in a manual index.
*
* It returns a Stream of Records, where records are
* specified per procedure. This particular procedure returns
* a stream of {@link SearchHit} records.
*
* The arguments to this procedure are annotated with the
* {@link Name} annotation and define the position, name
* and type of arguments required to invoke this procedure.
```

```
* There is a limited set of types you can use for arguments,
* these are as follows:
*
* <ul>
* <li>{@link String}</li>
* <li>{@link Long} or {@code long}</li>
* <li>{@link Double} or {@code double}</li>
* <li>{@link Number}</li>
* <li>{@link Boolean} or {@code boolean}</li>
*  <li>{@link java.util.Map} with key {@link String} and value {@link
Object}</li>
*  <li>{@link java.util.List} of elements of any valid argument type,
including {@link
java.util.List}</li>
* <li>{@link Object}, meaning any of the valid argument types</li>
* </ul>
*
* @param label the label name to query by
* @param query the lucene query, for instance `name:Brook*` to
* search by property `name` and find any value starting
263
* with `Brook`. Please refer to the Lucene Query Parser
* documentation for full available syntax.
* @return the nodes found by the query
*/
@Procedure("example.search")
@PerformsWrites
public Stream<SearchHit> search( @Name("label") String label,
@Name("query") String query )
{
String index = indexName( label );
// Avoid creating the index, if it's not there we won't be
// finding anything anyway!
if( !db.index().existsForNodes( index ))
{
// Just to show how you'd do logging
log.debug( "Skipping index query since index does not exist: `%s`", index );
return Stream.empty();
}
// If there is an index, do a lookup and convert the result
// to our output record.
return db.index()
.forNodes( index )
.query( query )
.stream()
.map( SearchHit::new );
}
/**
* This is the second procedure defined in this class, it is used to update
the
```

```
* index with nodes that should be queryable. You can send the same node
multiple
* times, if it already exists in the index the index will be updated to
match
* the current state of the node.
*
* This procedure works largely the same as {@link #search(String, String)},
*  with two notable differences. One, it is annotated with {@link
PerformsWrites},
* which is <i>required</i> if you want to perform updates to the graph in
your
* procedure.
*
* Two, it returns {@code void} rather than a stream. This is simply a short-
hand
* for saying our procedure always returns an empty stream of empty records.
*
* @param nodeId the id of the node to index
* @param propKeys a list of property keys to index, only the ones the node
* actually contains will be added
*/
@Procedure("example.index")
@PerformsWrites
public void index( @Name("nodeId") long nodeId,
@Name("properties") List<String> propKeys )
{
Node node = db.getNodeById( nodeId );
// Load all properties for the node once and in bulk,
// the resulting set will only contain those properties in `propKeys`
// that the node actually contains.
Set<Map.Entry<String,Object>> properties =
node.getProperties( propKeys.toArray( new String[0] ) ).entrySet();
// Index every label (this is just as an example, we could filter which
labels to index)
for ( Label label : node.getLabels() )
{
Index<Node> index = db.index().forNodes( indexName( label.name() ),
FULL_TEXT );
// In case the node is indexed before, remove all occurrences of it so
// we don't get old or duplicated data
index.remove( node );
// And then index all the properties
for ( Map.Entry<String,Object> property : properties )
{
index.add( node, property.getKey(), property.getValue() );
}
}
}
/**
* This is the output record for our search procedure. All procedures
```

```
264
* that return results return them as a Stream of Records, where the
* records are defined like this one - customized to fit what the procedure
* is returning.
*
* These classes can only have public non-final fields, and the fields must
* be one of the following types:
*
* <ul>
* <li>{@link String}</li>
* <li>{@link Long} or {@code long}</li>
* <li>{@link Double} or {@code double}</li>
* <li>{@link Number}</li>
* <li>{@link Boolean} or {@code boolean}</li>
* <li>{@link org.neo4j.graphdb.Node}</li>
* <li>{@link org.neo4j.graphdb.Relationship}</li>
* <li>{@link org.neo4j.graphdb.Path}</li>
*  <li>{@link  java.util.Map}  with  key  {@link  String}  and  value  {@link
Object}</li>
* <li>{@link java.util.List} of elements of any valid field type, including
{@link
java.util.List}</li>
* <li>{@link Object}, meaning any of the valid field types</li>
* </ul>
*/
public static class SearchHit
{
// This records contain a single field named 'nodeId'
public long nodeId;
public SearchHit( Node node )
{
this.nodeId = node.getId();
}
}
private String indexName( String label )
{
return "label-" + label;
}
}
```

4.2.6 事务管理

为了完全保持数据完整性并确保良好的事务行为，Neo4j 支持 ACID——原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）4 个要素。

- 原子性：如果事务的任何部分失败，整个事务不做任何操作，数据库状态保持不变。
- 一致性：任何事务都会使数据处于完全一致状态。
- 隔离性：在事务执行期间，被修改的数据无法由其他操作访问。
- 持久性：DBMS 可以恢复事务已提交了的结果。

特别需要注意如下几点：

（1）访问图、索引或模式的所有数据库操作都必须在事务中执行。
（2）默认隔离级别为 READ_COMMITTED。
（3）通过遍历检索的数据不受其他事务的修改保护。
（4）可以只有写锁被获取并保持，直到事务结束。
（5）可以手动获取节点和关系上的写锁，以实现更高级别的隔离（SERIALIZABLE）。
（6）在节点和关系级别都可以获取锁定权限。
（7）死锁检测被构建在核心事务管理中。

4.2.6.1　交互周期

访问图、索引或模式的所有数据库操作都必须在事务中执行。事务是被限制在线程内的，可以嵌套为“平行嵌套事务”。平行嵌套事务指的是所有嵌套事务都被添加到顶层事务的作用域中。嵌套事务可以标记顶层事务以进行回滚，这意味着整个事务都将被回滚，仅回滚在嵌套事务中的操作是不可能的。

处理事务的交互周期有如下所示 4 个步骤：

（1）开始一个事务。
（2）执行数据库操作。
（3）将事务标记为成功或不成功。
（4）完成事务。

完成每个事务是非常重要的。除非事务完成否则不会释放它已经获得的锁或内存。程序中惯用方法是使用 try-finally 块处理事务、启动事务，然后尝试执行图操作。在 try 块中的最后一个操作应该标记事务为成功即执行 success()方法，而 finally 块应该写完成事务的相关操作。完成事务将根据成功状态执行提交或回滚。

在事务中执行的所有修改都被暂时保存在内存中，所以应当将大型事务拆分为小型事务，以避免内存不足。

在使用线程池的情况下，当无法正确完成事务时，可能会发生其他错误。假设如果有一个没有正确完成的事务，它是被绑定到一个线程上的，当该线程被调度开始执行一个新的顶级事务时，它实际上将是一个嵌套事务。如果失败的事务状态是“标记为回滚”（如果检测到死锁，则会发生），则无法对该事务执行更多的工作。如果尝试这样做将导致在每次调用写操作时出错。

4.2.6.2　隔离级别

Neo4j 中的事务使用读提交隔离级别，也就是一旦提交事务就会马上看到数据的变更，并且不会在其他事务中看到尚未提交的数据。这种类型的隔离比串行化弱，但是这样显著提高了数据库的性能。

此外，Neo4j Java API 支持显式锁定节点和关系。使用锁可以通过明确获取和释放锁来模拟更高级别的隔离。例如，如果在公共节点或关系上进行写锁定，则所有事务将在该锁上串行化执行。

1. Cypher 中的更新丢失

Cypher 中，在某些情况下可以获取写锁来模拟改进的隔离。例如，多个并发 Cypher 查询增加属性值的情况。由于读提交隔离级别的限制，增加操作的最终结果可能无法确定。如果存在直接依赖，则 Cypher 将在读取前自动获取写锁定。直接依赖关系是指 SET 的右侧在表达式中读取依赖属性，或者在 map 字面值的键值对的值中。

例如，以下查询（如果由 100 个并发客户端运行）很可能不会将属性 n.prop 增加到 100，除非在读取属性值之前获取了写锁定。这是因为所有查询都会在自己的事务中读取 n.prop 的值，并且不会从尚未提交的任何其他事务中看到增加后的值。在最坏的情况下，如果所有线程在任何提交它们的事务之前执行读取，最终得到的值可能为 1。

【程序 4-31】Cypher 自动获取一个写锁定

```
MATCH (n:X {id: 42})
SET n.prop = n.prop + 1
MATCH (n)
SET n += { prop: n.prop + 1 }
```

有些情况下判断读写依赖过于复杂，所以在某些情况下 Cypher 不会自动加上写锁定，主要有如下两种情况：

（1）在读取请求的值之前，通过写入虚拟属性来获取节点的写锁定。

```
MATCH (n)
WITH n.prop as p
// ... 下面操作依赖变量 p 来产生 k 的值
SET n.prop = k + 1
```

（2）在同一查询中读取和写入的属性之间的循环依赖性。

```
MATCH (n)
SET n += { propA: n.propB + 1, propB: n.propA + 1 }
```

为了在复杂情况下确保行为的确定性，有必要在所操作的节点上显式获取写锁定。虽然在 Cypher 中没有明确的支持，但是可以通过写一个临时属性来解决这个限制就可以了。

如下例，在读取请求的值之前，通过写入虚拟属性来获取节点的写锁定。

```
MATCH (n:X {id: 42})
SET n._LOCK_ = true
WITH n.prop as p
// 下面操作依赖变量 p 来产生 k 的值
SET n.prop = k + 1
REMOVE n._LOCK_
```

在读取 n.prop 之前 SET n. _LOCK_语句的存在将确保在读取操作之前获取锁定，并且由于又对该节点上的所有并发查询进行了强制序列化，因此不会存在丢失更新的情况。

2. 默认自动加锁的情况

（1）在添加、更改或删除节点或关系上的属性时，将对操作的节点或关系执行写锁定。

（2）在创建或删除节点时，将为操作的节点执行写锁定。

（3）在创建或删除关系时，将对操作的关系及其两个节点执行写锁定。

锁将添加到事务中，并在事务完成时释放。

4.2.6.3　死锁

对于死锁需要了解以下内容：

在任何系统中如果使用锁，那么就可能发生死锁。然而，在 Neo4j 中在它们发生并抛出异常之前会检测任何死锁。在抛出异常之前，事务被标记为回滚。由事务获取的所有锁仍然被保留着，并将在事务完成时释放（在前面指出的 finally 块中）。一旦锁被释放，则因为等待导致死锁的事务而被锁住的其他事务得以继续执行。因导致死锁的事务而未执行的工作，在这之后就可以由用户在需要时重试。

有时候一些频繁的死锁是由发写请求的操作导致的，即不可能在同时满足预期的隔离和一致性的情况下执行它们，其解决方案是确保合理地并发更新。例如给定两个特定节点（A 和 B），当有两个或多个事务同时进行时，对于每个事务以随机顺序添加或删除这两个节点上的关系将导致死锁。一种解决方案是确保更新始终以相同的顺序发生（首先 A 然后 B）。另一种解决方案是确保每个线程/事务对某个节点或关系在并发事务中没有任何写入冲突。

1. 死锁处理程序

下面是如何在过程、服务器扩展或使用嵌入式 Neo4j 时处理死锁的示例。

用于代码片段的完整源代码可以在 GitHub 上源文件的 DeadlockDocTest.java 中找到。

当在代码中处理死锁时，有几个问题可能需要解决：

（1）需要进行有限的重试次数，如果达到阈值则失败。

（2）在每次尝试之间暂停一下，以允许其他事务完成，然后再次尝试。

（3）重试循环不仅可用于死锁，而且可用于其他类型的瞬态错误。

在以下部分中，将看到 Java 中的示例代码，其中显示了如何实现该示例代码。

使用 TransactionTemplate 处理死锁：如果不想自己写所有的代码，有一个类名为 TransactionTemplate，它将帮助我们实现所需要的处理。以下是如何创建、自定义和使用此模板以在事务中重试的示例。

首先，定义基本模板：

```
TransactionTemplate template = new
TransactionTemplate( ).retries( 5 ).backoff( 3, TimeUnit.SECONDS );
```

接下来，指定要使用的数据库和要执行的函数：

```
Object result = template.with(graphDatabaseService).execute( transaction -> {
    Object result1 = null;
    return result1;
} );
```

TransactionTemplate 使用 API 进行配置，可以选择是立即设置所有内容，还是在使用之前提

供一些详细信息。模板允许为重试的异常设置参数，并且还能够很容易地监视发生的事件。

2. 使用循环重试处理死锁

如果想回滚自己的循环重试代码，请参阅下面的代码，以下是循环重试的示例：

【程序 4-32】循环重试处理死锁

```
Throwable txEx = null;
int RETRIES = 5;
int BACKOFF = 3000;
for ( int i = 0; i < RETRIES; i++ )
{
    try ( Transaction tx = graphDatabaseService.beginTx() )
    {
        Object result = doStuff(tx);
        tx.success();
        return result;
    }
    catch ( Throwable ex )
    {
        txEx = ex;
        // 在此处添加想循环重试的内容
        if ( !(ex instanceof DeadlockDetectedException) )
        {
            break;
        }
    }
    // Wait so that we don't immediately get into the same deadlock
    if ( i < RETRIES - 1 )
    {
        try
        {
            Thread.sleep( BACKOFF );
        }
        catch ( InterruptedException e )
        {
            throw new TransactionFailureException( "Interrupted", e );
        }
    }
}
if ( txEx instanceof TransactionFailureException )
{
    throw ((TransactionFailureException) txEx);
}
else if ( txEx instanceof Error )
{
    throw ((Error) txEx);
}
else if ( txEx instanceof RuntimeException )
{
```

```
    throw ((RuntimeException) txEx);
}
else
{
    throw new TransactionFailureException( "Failed", txEx );
}
```

上面的循环重试代码可以按照需要重新定制。

3. 删除语义

在删除节点或者关系的时候，依附在节点或关系上的属性也会被一并删除，但是删除节点并不会删除指向此节点的关系，必须先删除指向此节点的关系才行。

在 Neo4j 中有一种强制的约束，就是任何关系必须有开始和结束节点，如果尝试删除一个仍被关系关联着的节点时，删除节点操作将会抛出异常而终止。所以必须先删除与节点关联的所有关系最后才能删除此节点，这个先后顺序必须遵守。

当在做删除操作时，我们必须了解如下一些概念：

（1）在删除节点或者关系的时候，依附在节点或关系上的属性也会被一并删除。

（2）在删除节点操作所在的事务提交时，此节点不能还有任何相关联的关系。

（3）在事务中如果含有删除节点或关系的操作，只要此事务没有被提交，那么程序中依然可以读取要删除的节点或关系，因为事务没被提交，它们还依然存在。

（4）在事务中如果已经运行了删除节点或关系的操作的代码，当此事务还没有被提交，如果继续尝试写或修改这些节点或关系，那么程序会抛出异常。

（5）如果事务中存在读取已经被删除了的节点或关系的操作，那么在事务提交时程序将会抛出异常。

4.2.6.4　创建唯一节点

如同传统关系型数据库，有些时候需要保证实体的唯一性。同样在 Neo4j 中也希望有一种机制可以确保创建的节点的唯一性，Neo4j 提供了三个主要的策略来确保唯一性，即单线程策略、获取或创建策略、消极锁策略。这三种策略可以运行在高可用性集群环境和单实例环境。

1. 单线程策略

使用单线程策略，系统中就不会同时存在多个线程去创建同一个实体（如节点、关系）。同样，在高可用性集群环境下，外部的单线程也可以在高可用性集群上执行操作并能确保唯一性。

2. 获取或创建策略

获取或创建唯一节点的首选方法是使用唯一性约束和 Cypher。有关更多信息，请参见“第 3 章 Neo4j 之 Cypher”使用 Cypher 和唯一约束获取或创建唯一节点。

通过使用 put-if-absent 功能，可以使用手动索引来保证实体唯一性。此时，手动索引将是一个锁，并且这个锁仅仅锁定用来保证线程和事务唯一性的最小资源。

3. 消极锁策略

虽然消极锁策略也是一个保证唯一性的策略，但 Neo4j 更推荐使用以上两个策略，如果都不可行，再考虑使用消极锁策略。

通过使用显式的、消极锁定，可以在多线程环境中实现实体创建的唯一性。通常情况下这个策略用于锁定单个或者一组公共的节点。

4. 事务事件

可以注册事务事件处理程序以接收 Neo4j 的事务事件。一旦在 GraphDatabaseService 实例中注册，它就会在提交事务之前接收事务的事件。当执行任何写操作并提交其事务后，处理程序会接收到通知。如果其事务的 success()方法没有被调用或事务被标记为失败即调用了 failure()方法，那么它将被回滚，并且没有任何事件被发送到处理程序。

在事务被提交之前，处理器的 beforeCommit 方法将被用在此事务中。此时，事务仍在运行，因此仍可进行更改。该方法还可以抛出异常，这将防止事务被提交。如果事务回滚，则将调用处理程序的 afterRollback 方法。

处理程序执行的顺序是未定义的，也就是说不能保证一个处理程序所做的更改将被其他处理程序看到。

如果在所有注册的处理程序中成功执行 beforeCommit 方法，那么将提交事务并使用相同的事务数据当作参数来调用 afterCommit 方法，并且此调用还包括从 beforeCommit 返回的对象。

在 afterCommit 中，事务已经关闭，并且访问 TransactionData 外部的任何内容都需要再打开一个新的事务。TransactionEventHandler 可以获得关于具有可通过 TransactionData 访问的任何更改的事务的通知，因此一些索引和模式更改不会触发这些事件。

4.2.7 使用 Java 在线备份 Neo4j

为了从基于 JVM 的开发平台上以编程方式备份完整或后续增量的数据，需要编写如下的 Java 代码：

【程序 4-33】在线备份数据库

```
OnlineBackup backup = OnlineBackup.from( "127.0.0.1" );
backup.full( backupPath.getPath() );
assertTrue( "Should be consistent", backup.isConsistent() );
backup.incremental( backupPath.getPath() );
```

4.2.8 使用 JMX 监控 Neo4j

为了能够连续地了解 Neo4j 数据库的运行状况，有不同级别的监控方法可用。这些方法大多数是基于 JMX（Java Management Extensions，即 Java 管理扩展）的。Neo4j 企业版还能够自动向常用的监控系统报告运行指标。

JMX 是一个为应用程序、设备、系统等植入管理功能的框架。JMX 可以跨越一系列异构操作系统平台、系统体系结构和网络传输协议，灵活地开发无缝集成的系统、网络和服务来管理应用。

4.2.8.1　使用 JMX 接入 Neo4j

默认情况下，Neo4j Enterprise Server 版本不允许远程 JMX 连接，因为 conf/neo4j.conf 配置文件中的相关选项已注释掉。要启用此功能，必须在配置文件中将 com.sun.management.jmxremote 选项取消注释。

有关 JMX 连接配置的详细信息，读者可以查询相关的 Java 文档。在此要提醒读者的是我们需要确保 JMX 的配置文件 conf/jmx.password 具有正确的文件权限。该文件的所有者必须是将运行服务的用户，并且权限应该是只读。在 UNIX 系统上就是 0600 权限。

在 Windows 系统上，我们可以按照 Oracle 在线文档[15]上的教程来设置正确的权限。如果在本地系统账户下运行服务，拥有该文件并且可以访问该文件的用户应该是 SYSTEM。

使用此设置，应该能够在服务器 IP 地址的 3637 端口来连接到 Neo4j 服务器的 JMX 监视器，当然我们需要提供用户名和密码。

4.2.8.2　使用 JMX 和 JConsole 接入 Neo4j 实例

首先，通过命令启动 Neo4j。

```
$NEO4j_HOME/bin/neo4j start
```

然后，使用命令启动 jconsole。

```
$JAVA_HOME/bin/jconsole
```

在图 4-9 所示的控制面板中，选择链接到运行 neo4j 的进程。

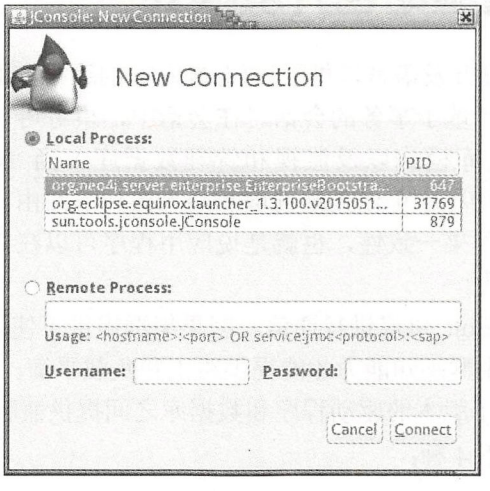


图 4-9　jconsole 选择链接 neo4j 进程

现在，在 MBeans 选项卡中看到一个 org.neo4j 节点列表。在此列表下，将可以查看 Neo4j 所有公开的监控信息，如图 4-10 所示。

[15] http://docs.oracle.com/javase/8/docs/technotes/guides/management/security-windows.html

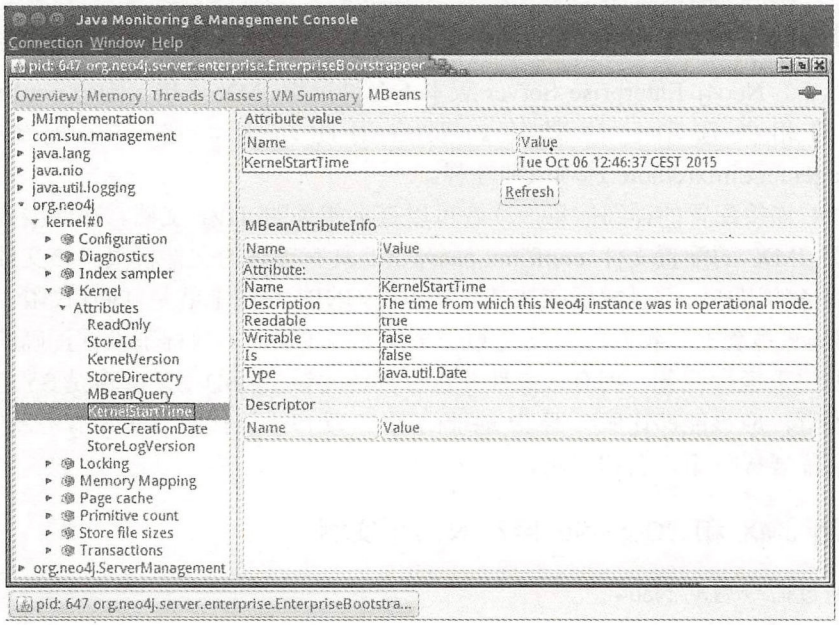


图 4-10　jconsoleNeo4j 公开的所有监控信息

4.3 各语言驱动包开发模式

Neo4j 驱动程序为其他开发语言提供了访问 Neo4j 的途径，编写使用驱动程序的应用程序实例就可以与数据库进行基于事务的会话。在会话中，事务可以运行创建和查询数据的相关命令，也可以定义数据库模式以及监视和管理数据库。当 Neo4j 部署在因果（Causal Clustering）集群中时，驱动程序可以处理读取和写入操作的路由。使用因果集群驱动程序还提供书签，书签用于确保因果一致性，也就是说应用程序可以在集群的不同成员上运行多个查询，同时保持数据的一致。

Neo4j 驱动程序使用 Bolt 协议进行通信。因果集群提供了使用驱动程序获取集群拓扑的功能。驱动程序必须具有集群感知能力才能提供路由和负载平衡。

最后，使用 Bolt 在多个版本的驱动程序和数据库之间提供兼容性。

官方驱动程序包括以下 4 种：

- .NET
- Java
- JavaScript
- Python

Neo4j 驱动程序也可以在任何不是基于 JVM 的语言中使用，包括.NET、JavaScript、Python 等。.NET 驱动程序用 C#编写，可以使用任何.NET 语言，包括 C#、F#和 VB，甚

仅供非商业用途或交流学习使用

至可以直接从 PowerShell 中运行。

4.3.1　驱动开发入门

表 4-4 显示了 Neo4j 版本与驱动程序版本的对应关系。

表 4-4　驱动程序 Neo4j 对应版本号

驱动版本号	Neo4j 版本号	Bolt 协议版本号
1.1	3.1	1
1.1	3.0<*>	1
1.0	3.1<*>	1
1.0	3.0	1

驱动程序版本 1.0 和 1.1 可以与 Neo4j 3.0 和 3.1 一起配合使用。要利用 Neo4j 3.1 中的因果集群模式并使用诸如书签和路由之类的功能，必须使用 1.1 版本的驱动程序。

1. 使用依赖管理获得驱动

可以下载驱动程序源或使用语言的依赖关系管理器获取驱动程序包。

C#：

在 Visual Studio 包管理控制台使用 NuGet 命令：

```
PM> Install-Package Neo4j.Driver -Version 1.1.0-M02
```

Java：

使用 Maven 依赖管理，在 pom.xml 文件中添加如下命令：

```
<dependencies>
    <dependency>
        <groupId>org.neo4j.driver</groupId>
        <artifactId>neo4j-java-driver</artifactId>
        <version>1.1.0</version>
    </dependency>
</dependencies>
```

对于 Gradle 或者 Grails，使用如下命令：

```
compile 'org.neo4j.driver:neo4j-java-driver:1.1.0'
```

JavaScript：

使用 NPM 包管理，命令如下：

```
npm install neo4j-driver@1.1.0-M03
```

Python：

使用 pip 包管理，命令如下：

```
pip install neo4j-driver==1.1.0b1
```


仅供非商业用途或交流学习使用

2. 使用官方驱动包

每个 Neo4j 驱动程序都有一个用于创建驱动程序的数据库对象。要使用驱动程序，请按照以下步骤：

（1）向数据库对象请求一个新的驱动程序。
（2）向驱动程序对象请求一个新会话。
（3）请求会话对象创建事务。
（4）使用事务对象运行语句。它返回一个表示结果的对象。
（5）处理结果。
（6）关闭会话。

C#

首先使用 using 引入命名空间。

```
using Neo4j.Driver.V1;
```

然后就可以用 C#嵌入 Cypher 操作数据库了。

【程序 4-34】C#嵌入 Cypher 代码

```
using (var driver = GraphDatabase.Driver("bolt://localhost",
AuthTokens.Basic("neo4j", "neo4j")))
using (var session = driver.Session())
{
    session.Run("CREATE (a:Person {name:'Arthur', title:'King'})");
    var result = session.Run("MATCH (a:Person) WHERE a.name = 'Arthur' RETURN
a.name AS name, a.title AS title");
    foreach (var record in result)
    {
        Output.WriteLine($"{record["title"].As<string>()}
{record["name"].As<string>()}");
    }
}
```

Java：

首先使用 import 引入包。

```
import org.neo4j.driver.v1.AuthTokens;
import org.neo4j.driver.v1.Driver;
import org.neo4j.driver.v1.GraphDatabase;
import org.neo4j.driver.v1.Record;
import org.neo4j.driver.v1.Session;
import org.neo4j.driver.v1.StatementResult;
import org.neo4j.driver.v1.Transaction;
import static org.neo4j.driver.v1.Values.parameters;
```

Java 嵌入 Cypher 的代码如下：

仅供非商业用途或交流学习使用

【程序 4-35】Java 嵌入 Cypher 代码

```
Driver driver = GraphDatabase.driver( "bolt://localhost:7687",
AuthTokens.basic( "neo4j", "neo4j" ) );
try ( Session session = driver.session() )
{
    try ( Transaction tx = session.beginTransaction() )
    {
        tx.run( "CREATE (a:Person {name: {name}, title: {title}})",
                parameters( "name", "Arthur", "title", "King" ) );
        tx.success();
    }
    try ( Transaction tx = session.beginTransaction() )
    {
        StatementResult result = tx.run( "MATCH (a:Person) WHERE a.name =
{name} " +"RETURN a.name AS name, a.title AS title",
                                        parameters( "name", "Arthur" ) );
        while ( result.hasNext() )
        {
            Record record = result.next();
            System.out.println( String.format( "%s %s",
record.get( "title" ).asString(), record.get( "name" ).asString() ) );
        }
    }
}
driver.close();
```

JavaScript:

引入驱动程序对象

```
var neo4j = require('neo4j-driver').v1;
```

JavaScript 嵌入 Cypher 的代码如下：

【程序 4-36】JavaScript 嵌入 Cypher 代码

```
var driver = neo4j.driver("bolt://localhost", neo4j.auth.basic("neo4j",
"neo4j"));
var session = driver.session();
session
  .run( "CREATE (a:Person {name:'Arthur', title:'King'})" )
  .then( function()
  {
    return session.run( "MATCH (a:Person) WHERE a.name = 'Arthur' RETURN
a.name AS name, a.title AS title" )
  })
  .then( function( result ) {
    console.log( result.records[0].get("title") + " " +
result.records[0].get("name") );
    session.close();
    driver.close();
```


仅供非商业用途或交流学习使用

```
})
```

Python：

引入驱动程序对象：

```
from neo4j.v1 import GraphDatabase, basic_auth
```

Python 嵌入 Cypher 的代码如下：

【程序 4-37】Python 嵌入 Cypher 代码

```
driver = GraphDatabase.driver("bolt://localhost", auth=basic_auth("neo4j",
"neo4j"))
session = driver.session()
session.run("CREATE (a:Person {name:'Arthur', title:'King'})")
result = session.run("MATCH (a:Person) WHERE a.name = 'Arthur' RETURN a.name
AS name, a.title AS title")
for record in result:
   print("%s %s" % (record["title"], record["name"]))
session.close()
```

也可以直接在会话 session 中运行语句，然后 session 将负责打开和关闭事务。这对于单独的只读查询很方便。

4.3.2 配置和连接

接下来讨论如何配置驱动程序并使它连接到 Neo4j 数据库。驱动程序提供 session，session 封装了针对数据库的一个或多个工作事务的上下文。

驱动程序对象实例可以在整个应用程序中跨多个线程进行全局共享，它包含一个到集群的各个成员的连接池，可以借用它来执行任何操作。

关闭驱动程序将关闭该驱动程序生成的所有连接，无论它们当前是否正在使用。

4.3.2.1 部署

驱动程序包括一个图数据库对象。此对象提供驱动程序实例。当从数据库对象请求驱动程序实例时，需要提供 URI。URI 声明了 Neo4j 服务器的协议、主机和端口。

有两种 Bolt URI：“直接 URI”和“路由 URI”。直接 URI 用于连接到单个实例，如果应用程序依赖于路由和负载平衡，则必须单独提供。驱动程序不知道它可能连接到的任何集群，只能直接连接到单个服务器。此模式适用于使用独立实例或部署自定义负载均衡的情况（例如使用 Neo4j Highly Availability 集群和 HAProxy）。

路由 URI 用于创建支持集群的驱动程序。路由驱动程序连接到集群并处理路由和负载均衡。在为驱动程序提供 URI 之后，它将连接并请求以获得集群拓扑的描述信息。然后驱动程序能够选择最适合的集群成员创建会话。这种模式适合使用 Neo4j 因果集群。

Bolt URI 格式

Bolt URI 遵循 scheme://host:port 的标准 URI 模式。例如，bolt://203.0.113.1:1234 将使用

仅供非商业用途或交流学习使用

Bolt 协议连接到 203.0.113.1 的 1234 端口上。可以使用主机名代替 IP 地址：bolt://server:1234。默认端口为 7687。

路由 URI 使用方案 bolt://server:1234。

下面展示如何在代码中利用直接 URI 创建驱动实例：

C#：

```
var driver = GraphDatabase.Driver("bolt://localhost",
AuthTokens.Basic("neo4j", "neo4j"));
```

Java：

```
Driver driver = GraphDatabase.driver( "bolt://localhost:7687",
AuthTokens.basic( "neo4j", "neo4j" ) );
```

JavaScript：

```
var driver = neo4j.driver("bolt://localhost", neo4j.auth.basic("neo4j",
"neo4j"));
```

Python：

```
driver = GraphDatabase.driver("bolt://localhost", auth=basic_auth("neo4j",
"neo4j"))
```

路由 URI 用来创建用于使用 Neo4j 因果集群的路由驱动程序。

下面例子是用路由 URI 创建驱动的实例：

```
Driver driver = GraphDatabase.driver( "bolt+routing://localhost",
AuthTokens.basic("neo4j", "neo4j") );
```

创建驱动程序应用时需要传递一个 URI 当作初始化参数来连接服务器，如果没有服务器可用于在该 URI 响应，则抛出 ServiceUnavailableException。因此，在应用实例中可以组装一个 URI 的 list 循环创建连接，这样可以比较可靠。

下面方法可以使用多个路由 URI 创建驱动实例：

【程序 4-38】多个路由 URI 创建驱动实例

```
public Driver acquireDriver(List<String> uris, AuthToken authToken, Config
config)
{
    for (String uri : uris)
    {
        try {
            return GraphDatabase.driver(uri, authToken, config);
        }
        catch (ServiceUnavailableException ex)
        {
            // This URI failed, so loop around again if we have another.
        }
    }
    throw new ServiceUnavailableException("No valid database URI found");
}
```


仅供非商业用途或交流学习使用

对于某些配置，还得考虑在 Bolt URI 中使用主机域名而不是 IP 地址。这虽然多了一层地址解析操作，但在 DNS 域名解析层可以设置备用服务器的重定向，这样增加了程序的灵活性。

4.3.2.2 配置

驱动程序中的配置设置可以给驱动程序提供构造函数的参数。在接下来的部分将介绍如何具体操作。

1. 加密和授权

所有 Neo4j 驱动程序支持使用 SSL / TLS 对 Neo4j 驱动程序和 Neo4j 实例之间的连接进行加密。

Neo4j 默认情况下只允许对连接加密，但在使用本地地址 localhost 时，不会启用加密，因为本地连接在开发环境中常用于测试。通常在多服务器的情况下，多个数据库服务器一个接一个地启动和关闭，这样使用自动生成的证书和“首先使用信任”信任策略是有问题的。由第一个服务器生成的证书与 Bolt URI 相关联，如果下一个启动的服务器使用相同的 Bolt URI 连接，则证书将不匹配，并且连接将失败。

为了解决这个问题，默认配置使用 ENCRYPTION_NON_LOCAL 策略。这意味着默认情况下与远程主机的通信是加密的，但本地连接是不使用加密的。

当然也可以修改配置，对所有连接都进行加密。有关详细信息，请参阅“5.1.7 配置 Neo4j 连接器”小节。

强烈建议使用加密功能以保护 Neo4j 中存储的身份验证凭据和数据。

由于技术限制，.NET 驱动程序默认情况下未启用加密。默认情况下，当在 Web 浏览器中运行时，JavaScript 驱动程序是没有启用加密的。

2. 加密

驱动程序可以通过配置来打开或关闭 TLS 加密。建议只应在可信的内部网络中关闭加密。虽然大多数驱动程序默认启用加密，但最好是明确配置为使用加密。这样在开发人员写程序时就不会过度依赖数据库的默认配置。

下面代码是配置驱动程序启用 TLS 加密：

C#：

```
driver = GraphDatabase.driver("bolt://localhost", auth=basic_auth("neo4j",
"neo4j"), encrypted=True)
```

Java：

```
Driver driver = GraphDatabase.driver( "bolt://localhost:7687",
AuthTokens.basic( "neo4j", "neo4j" ),
Config.build().withEncryptionLevel( Config.EncryptionLevel.REQUIRED ).
toConfig() );
```

JavaScript：

```
var driver = neo4j.driver("bolt://localhost", neo4j.auth.basic("neo4j",
```


仅供非商业用途或交流学习使用

```
"neo4j"), {
  //在 NodeJS 中，默认情况下加密是 ENCRYPTION_NON_LOCAL。在 Web 软件包中，它是
ENCRYPTION_OFF。
  encrypted:"ENCRYPTION_ON"
});
```

3. 解密

当建立加密连接时，需要验证远程对等端是期望连接到的对等端。没有验证这一点，黑客可以容易地执行所谓的“中间人”攻击，欺骗获取加密连接。因此，Neo4j 驱动程序不提供建立加密连接的方式。

有两种信任策略：

- 信任第一次使用策略。
- 信任签名的证书策略。

信任第一次使用策略意味着驱动程序将信任对主机的第一个连接是安全和有效的。在后续连接上，驱动程序将验证它与第一个连接上的主机通信。为了确保此身份验证策略有效，必须在可信网络环境中建立与服务器的第一个连接。

此策略与 ssh 命令行工具使用的默认策略相同。它提供了良好的安全性。因此，这个是所有平台上的默认策略。

由于技术限制，此策略不适用于.NET 平台或在 Web 浏览器中使用 JavaScript 驱动程序。

下面代码是配置驱动信任机制的代码，C# 版本不支持，所以只有 Java、JavaScript、Python 版本：

Java：

```
Driver driver =
      GraphDatabase.driver( "bolt://localhost:7687",
AuthTokens.basic( "neo4j", "neo4j" ), Config.build()
            .withEncryptionLevel( Config.EncryptionLevel.REQUIRED )
            .withTrustStrategy(
                  Config.TrustStrategy.trustOnFirstUse( new
File( "/path/to/neo4j_known_hosts" ) ) )
            .toConfig() );
```

JavaScript：

```
var driver = neo4j.driver("bolt://localhost", neo4j.auth.basic("neo4j",
"neo4j"), {
  // 注意 trust-on-first-use 在浏览器端是不支持的,
  // 在 NodeJS 中 trust-on-first-use 是默认的信任模式. 在浏览器中
  // 是 TRUST_CUSTOM_CA_SIGNED_CERTIFICATES.
  trust: "TRUST_ON_FIRST_USE",
  encrypted:"ENCRYPTION_NON_LOCAL"
});
```

Python：

```
driver = GraphDatabase.driver("bolt://localhost:7687",
auth=basic_auth("neotest", "neotest"), encrypted=True,
```


仅供非商业用途或交流学习使用

```
trust=TRUST_ON_FIRST_USE)
```

信任签名证书策略意味着驱动程序如果提供了驱动程序信任的证书，那么它将被允许连接到 Neo4j 服务器。所有驱动程序都支持这种信任的证书策略。

在不同驱动程序开发上安装受信任的证书使用的方法是不同的。Java 驱动程序直接传入可信证书的文件路径，并在运行时自动加载证书即可。.NET 和 Python 驱动程序需要首先将证书安装到操作系统的受信任证书库中。

下面代码是配置驱动程序以签名的证书文件为信任模式：

C#：

```
var driver = GraphDatabase.Driver("bolt://localhost",
AuthTokens.Basic("neo4j", "neo4j"),
Config.Builder.WithEncryptionLevel(EncryptionLevel.Encrypted).ToConfig());
```

Java：

```
Driver driver =
      GraphDatabase.driver( "bolt://localhost:7687",
AuthTokens.basic( "neo4j", "neo4j" ), Config.build()
            .withEncryptionLevel( Config.EncryptionLevel.REQUIRED )
            .withTrustStrategy( Config.TrustStrategy
                  .trustCustomCertificateSignedBy( new File( "/path/to/ca-
certificate.pem" ) ) )
            .toConfig() );
```

JavaScript：

```
var driver = neo4j.driver("bolt://localhost", neo4j.auth.basic("neo4j",
"neo4j"), {
  trust: "TRUST_CUSTOM_CA_SIGNED_CERTIFICATES",
  // 只在 NodeJS 中可以配置驱动使用签名证书文件的信任模式
  // 在浏览器中根据浏览器的不同，对签名证书的支持也不一样.
  trustedCertificates : ["path/to/ca.crt"],
  encrypted:"ENCRYPTION_NON_LOCAL"
});
```

Python：

```
driver = GraphDatabase.driver("bolt://localhost", auth=basic_auth("neo4j",
"neo4j"), encrypted=True, trust=TRUST_SIGNED_CERTIFICATES)
```

4. 客户端授权

服务器将要求驱动程序为用户提供身份验证凭证，以便连接到数据库，除非已禁用身份验证。

C#：

```
var driver = GraphDatabase.Driver("bolt://localhost",
Config.Builder.WithEncryptionLevel(EncryptionLevel.Encrypted).ToConfig());
```

Java：

```
Driver driver = GraphDatabase.driver( "bolt://localhost:7687",
Config.build().withEncryptionLevel( Config.EncryptionLevel.REQUIRED )
```


仅供非商业用途或交流学习使用

```
.toConfig() );
```

JavaScript:

```
var driver = neo4j.driver("bolt://localhost", {
  // 在 NodeJS 中默认是开启加密的，在浏览器中默认是关闭加密的
  encrypted:"ENCRYPTION_NON_LOCAL"
});
```

Python:

```
driver = GraphDatabase.driver("bolt://localhost", encrypted=True)
```

如果驱动程序和服务器之间的通信未加密，则认证凭证将作为纯文本发送。强烈建议在使用身份验证时使用加密。

4.3.3　执行 Cypher 语句

与 Neo4j 数据库的所有交互都在 session 中进行的。每个 session 会话是针对数据库的一个或多个事务。

session 会话是用于事务处理的轻量级容器。Session 可以从驱动程序对象中获取。

使用路由驱动程序时，可以将 session 会话访问模式声明为 READ 或 WRITE。驱动程序将针对所选工作模式对指定的集群成员创建会话。

在通常情况下，事务可以保证从用 session 创建运行的语句到对于每个语句的结果处理以及任何异常都能够在可控的范围内。当事务序列全部被执行完毕后，会话被关闭。

下面给出各个语言嵌入 Cypher 并执行的代码：

C#:

```
using (var tx = session.BeginTransaction())
{
    tx.Run("CREATE (:Person {name: 'Guinevere'})");
    tx.Success();
}
```

Java:

```
try ( Transaction tx = session.beginTransaction() )
{
    tx.run( "CREATE (:Person {name: {name}})",
            Values.parameters( "name", "Guinevere" ) );
    tx.success();
}
```

JavaScript:

```
var tx = session.beginTransaction();
tx.run( "CREATE (:Person {name: 'Guinevere'})" );
tx.commit().then(function() {session.close()});
```

Python:

```
with session.begin_transaction() as tx:
    tx.run("CREATE (:Person {name: 'Guinevere'})")
    tx.success = True
```


仅供非商业用途或交流学习使用

4.3.3.1 Session

Session 会话如同 Web 开发中的 Session，可以区分开不同的操作实例。可以使用 driver.session 方法来创建一个新的 Session。Session 会话创建是一个低资源消耗的过程。底层连接将从构建的驱动程序连接池创建 Session，并在关闭时返还到连接池。

要注意，Session 会话不是线程安全的。

1. 访问模式

当驱动程序启用路由时，需要先指定访问模式：READ 或 WRITE。如果省略，则默认使用 WRITE 模式。

路由驱动程序将分别针对适合于写入或读取的实例创建会话。针对因果集群进行部署意味着驱动程序可以有效地路由读取和写入。读取是针对只读副本，而写入集群的操作需要进行一致提交和保管处理。

如果创建会话的实例对于声明的访问模式不再可用，那么应用程序将接收到一个 SessionExpiredException。发生这种情况时，应用程序应从驱动程序获取另一个会话并重新执行失败的事务。

2. 连接池

驱动程序具有连接池，Session 会话将借用连接来执行其工作。当会话关闭时，Session 将返回到连接池。最重要的是确保会话是正确关闭的，以便结果和错误完全由客户端接收，并且 Session 连接可以重新使用。

另外，当抛出异常时 Session 会话会被关闭，这样可以确保执行所有操作是稳定可靠的。

4.3.3.2 回滚事务

在代码中想要回滚事务是非常容易的：

C#：

```
using (var tx = session.BeginTransaction())
{
    tx.Run("CREATE (:Person {name: {name}})",
        new Dictionary<string, object> {{"name", "Merlin"}});
    tx.Failure(); // This step is optional. If not called, tx.Failure() is
implicit.
}
```

Java：

```
try ( Transaction tx = session.beginTransaction() )
{
    tx.run( "CREATE (:Person {name: {name}})",
          Values.parameters( "name", "Merlin" ) );
    tx.failure();
}
```

JavaScript：

```
var tx = session.beginTransaction();
```

```
tx.run( "CREATE (:Person {name: 'Merlin'})" );
tx.rollback().then(function() {session.close()});
```

Python:

```
with session.begin_transaction() as tx:
    tx.run("CREATE (:Person {name: 'Merlin'})")
    tx.success = False
```

4.3.3.3　自动提交事务

虽然通常情况下事务在程序中是被明确声明了的，但有些情况下 Session 的 run 方法可以隐含地自动提交需要运行语句的事务，这只出现在只读查询的情况下，因为读操作不会改变数据库内容。

自动提交事务不支持书签功能，并且在调试时难以诊断故障。

下面例子是自动提交事务的代码：

C#:

```
var result = session.Run("CREATE (person:Person {name: {name}})",
    new Dictionary<string, object> {{"name", "Arthur"}});
```

Java:

```
StatementResult result =
      transaction.run( "CREATE (person:Person {name: {name}})",
Values.parameters( "name", "Arthur" ) );
transaction.success();
```

JavaScript:

```
session
  .run( "CREATE (person:Person {name: {name}})", {name: "Arthur"} )
```

Python:

```
result = session.run("CREATE (person:Person {name: {name}})", {"name":
"Arthur"})
```

4.3.3.4　查询语句中的参数

建议在查询语句中始终使用参数。使用参数具有显著的性能和安全优势：

- 参数允许查询中重复使用，使查询更加高效。
- 它保护数据库免受 Cypher 注入攻击。

而且，可以运行带有字面值（literal）的查询。

C#:

```
result = session.run("CREATE (person:Person {name: 'Arthur'})")
```

Java:

```
StatementResult result = transaction.run( "CREATE (p:Person { name:
'Arthur' })" );
transaction.success();
```

JavaScript：

```
session
  .run( "CREATE (p:Person { name: 'Arthur' })" )
```

Python：

```
result = session.run("CREATE (person:Person {name: 'Arthur'})")
```

4.3.3.5 书签（Bookmark）

可以在 Session 执行了某些操作后创建一个书签，书签是指向某些操作的指针。当为其他相关工作创建新会话时，可以使用已经创建的书签。书签确保之后的操作对依赖于先前的操作。无论数据是否已复制到集群的每个实例，书签都保证下一个操作会被应用于先前设置了标签的实例。

下面程序实例为写入集群创建一个表示工作的逻辑位置的书签。

【程序 4-39】创建书签

```
Driver driver = GraphDatabase.driver( seedUri, Authtokens.basic( "neo4j",
"neo4j" ) );
String bookmark;
try ( Session session = driver.session( AccessMode.WRITE ) )
{
    try ( Transaction tx = session.beginTransaction() )
    {
        tx.run( "CREATE (person:Person {name: {name}, title: {title}})",
              parameters( "name", "Arthur", "title", "king" ) );
        tx.success();
        tx.close();
    }
    finally
    {
        bookmark = session.lastBookmark();
    }
}
return bookmark;
```

创建书签后，在集群随后的操作中使用它。Session 会话将会为加过书签的实例提供服务。数据的视图对于应用程序是一致的。

【程序 4-40】使用书签

```
try ( Session session = driver.session( AccessMode.READ ) )
{
    StatementResult result;
    try ( Transaction tx = session.beginTransaction( bookmark ) )
    {
        result = tx.run( "MATCH (person:Person {name: {name}}) RETURN
person.title",
              parameters( "name", "Arthur" ) );
        tx.success();
```

```
        tx.close();
        while ( result.hasNext() )
        {
            Record record = result.next();
            System.out.println( "Arthur's title is " +
record.get( "person.title" ) );
        }
    }
}
```

4.3.4　返回结果

4.3.4.1　结果游标

结果游标提供了对结果记录流的访问。游标可以不断依次指向结果集合中的每个记录。在将游标移动到第一条记录之前，它会先指向记录的起始位置。一旦游标已经遍历到结果流的末尾，就可以使用摘要信息和元数据。

随着游标在结果集中前进，结果记录将被惰性加载。这意味着必须将游标移动到第一个结果，然后才能使用此结果。通常最好的做法是明确使用结果和关闭会话，特别是在运行 update 语句时。

下面代码是使用游标的示例：

C#:

```
var searchTerm = "Sword";
var result = session.Run("MATCH (weapon:Weapon) WHERE weapon.name CONTAINS
{term} RETURN weapon.name",
    new Dictionary<string, object> { { "term", searchTerm } });
Output.WriteLine($"List of weapons called {searchTerm}:");
foreach (var record in result)
{
    Output.WriteLine(record["weapon.name"].As<string>());
}
```

Java:

```
String searchTerm = "Sword";
try ( Transaction tx = session.beginTransaction() )
{
    StatementResult result =
            tx.run( "MATCH (weapon:Weapon) WHERE weapon.name CONTAINS {term}
RETURN weapon.name",
                    Values.parameters( "term", searchTerm ) );
    System.out.println( "List of weapons called " + searchTerm + ":" );
    while ( result.hasNext() )
    {
        Record record = result.next();
        System.out.println( record.get( "weapon.name" ).asString() );
    }
}
```

JavaScript:

```
var searchTerm = "Sword";
session
  .run( "MATCH (weapon:Weapon) WHERE weapon.name CONTAINS {term} RETURN
weapon.name", {term : searchTerm} )
  .subscribe({
    onNext: function(record) {
      console.log("" + record.get("weapon.name"));
    },
    onCompleted: function() {
      session.close();
    },
    onError: function(error) {
      console.log(error);
    }
  });
```

Python:

```
search_term = "Sword"
result = session.run("MATCH (weapon:Weapon) WHERE weapon.name CONTAINS
{term} "
                "RETURN weapon.name", {"term": search_term})
print("List of weapons called %r:" % search_term)
for record in result:
    print(record["weapon.name"])
```

结果记录集合由记录组成。记录提供结果的一部分固定不变的内容，它是键和值的有序映射。这些键值对称为字段，由于字段都是有键和有序的，因此可以通过位置（基于 0 的整数）或键（字符串）访问字段的值。

下面代码是使用记录的示例：

C#:

```
var searchTerm = "Arthur";
var result = session.Run("MATCH (weapon:Weapon) WHERE weapon.owner CONTAINS
{term} RETURN weapon.name, weapon.material, weapon.size",
    new Dictionary<string, object> { { "term", searchTerm } });
Output.WriteLine($"List of weapons owned by {searchTerm}:");
foreach (var record in result)
{
    var list = record.Keys.Select(key => $"{key}: {record[key]}").ToList();
    Output.WriteLine(string.Join(", ", list));
}
```

Java:

```
String searchTerm = "Arthur";
try ( Transaction tx = session.beginTransaction() )
{
    StatementResult result = tx.run(
            "MATCH (weapon:Weapon) WHERE weapon.owner CONTAINS {term} RETURN
weapon.name, weapon.material, " +
```

```
            "weapon.size",
            Values.parameters( "term", searchTerm ) );
    System.out.println( "List of weapons owned by " + searchTerm + ":" );
    while ( result.hasNext() )
    {
        Record record = result.next();
        List<String> sword = new ArrayList<>();
        for ( String key : record.keys() )
        {
            sword.add( key + ": " + record.get( key ) );
        }
        System.out.println( sword );
    }
}
```

JavaScript:

```
var searchTerm = "Arthur";
session
  .run( "MATCH (weapon:Weapon) WHERE weapon.owner CONTAINS {term} RETURN
weapon.name, weapon.material, weapon.size", {term : searchTerm} )
  .subscribe({
    onNext: function(record) {
      var sword = [];
      record.forEach(function(value, key)
      {
        sword.push(key + ": " + value);
      });
      console.log(sword);
    },
    onCompleted: function() {
      session.close();
    },
    onError: function(error) {
      console.log(error);
    }
  });
```

Python:

```
search_term = "Arthur"
result = session.run("MATCH (weapon:Weapon) WHERE weapon.owner CONTAINS
{term} "
                     "RETURN weapon.name, weapon.material, weapon.size",
{"term": search_term})
print("List of weapons owned by %r:" % search_term)
for record in result:
    print(", ".join("%s: %s" % (key, record[key]) for key in record.keys()))
```

4.3.4.2　保留结果

直到在 Session 会话中运行另一个语句之前或直到当前事务关闭结果集都可用。如果需

要保留特定的结果集，每个驱动程序提供了收集结果集合并将其转换为每种语言的标准数据结构的方法。保存的结果也可以直接用于其他待运行的新语句中。

下面代码是使用保留结果的示例：

C#:

```
var result = session.Run("MATCH (knight:Person:Knight) WHERE knight.castle =
{castle} RETURN knight.name AS name",
    new Dictionary<string, object> { { "castle", "Camelot" } });
var records = result.ToList();
session.Dispose();
foreach (var record in records)
{
    Output.WriteLine($"{record["name"].As<string>()} is a knight of
Camelot");
}
```

Java:

```
List<Record> records;
try ( Session session = driver.session() )
{
    try ( Transaction tx = session.beginTransaction() )
    {
        StatementResult result = tx.run(
                "MATCH (knight:Person:Knight) WHERE knight.castle = {castle}
RETURN knight.name AS name",
                Values.parameters( "castle", "Camelot" ) );
        records = result.list();
    }
}
for ( Record record : records )
{
    System.out.println( record.get( "name" ).asString() + " is a knight of
Camelot" );
}
```

JavaScript:

```
session
  .run("MATCH (knight:Person:Knight) WHERE knight.castle = {castle} RETURN
knight.name AS name", {castle: "Camelot"})
  .then(function (result) {
    var records = [];
    for (var i = 0; i < result.records.length; i++) {
      records.push(result.records[i]);
    }
    return records;
  })
  .then(function (records) {
    for(var i = 0; i < records.length; i ++) {
      console.log(records[i].get("name") + " is a knight of Camelot");
    }
```

```
    session.close();
  });
```

Python:

```
session = driver.session()
result = session.run("MATCH (knight:Person:Knight) WHERE knight.castle =
{castle} "
                     "RETURN knight.name AS name", {"castle": "Camelot"})
retained_result = list(result)
session.close()
for record in retained_result:
    print("%s is a knight of Camelot" % record["name"])
```

在查询中使用结果：

C#:

```
var result = session.Run("MATCH (knight:Person:Knight) WHERE knight.castle =
{castle} RETURN id(knight) AS knight_id",
    new Dictionary<string, object> { { "castle", "Camelot" } });
foreach (var record in result)
{
    session.Run("MATCH (knight) WHERE id(knight) = {id} " +
                "MATCH (king:Person) WHERE king.name = {king} " +
                "CREATE (knight)-[:DEFENDS]->(king)",
        new Dictionary<string, object> {{"id", record["knight_id"]}, {"king",
"Arthur"}});
}
```

Java:

```
StatementResult result = null;
try ( Transaction transaction = session.beginTransaction() )
{
    result = transaction.run(
            "MATCH (knight:Person:Knight) WHERE knight.castle = {castle} RETURN
id(knight) AS knight_id",
            Values.parameters( "castle", "Camelot" ) );
}
for ( Record record : result.list() )
{
    try ( Transaction tx = session.beginTransaction() )
    {
        tx.run( "MATCH (knight) WHERE id(knight) = {id} " +
                "MATCH (king:Person) WHERE king.name = {king} " +
                "CREATE (knight)-[:DEFENDS]->(king)",
                Values.parameters( "id", record.get( "knight_id" ), "king",
"Arthur" ) );
        tx.success();
    }
}
```

JavaScript:

```
session
  .run("MATCH (knight:Person:Knight) WHERE knight.castle = {castle} RETURN
id(knight) AS knight_id", {"castle": "Camelot"})
  .subscribe({
    onNext: function(record) {
      session
        .run("MATCH (knight) WHERE id(knight) = {id} MATCH (king:Person)
WHERE king.name = {king} CREATE (knight)-[:DEFENDS]->(king)",
        {"id": record.get("knight_id"), "king": "Arthur"});
    },
    onCompleted: function() {
      session
        .run("MATCH (:Knight)-[:DEFENDS]->() RETURN count(*)")
        .then(function (result) {
          console.log("Count is " + result.records[0].get(0).toInt());
          session.close();
        });
    },
    onError: function(error) {
      console.log(error);
    }
  });
```

Python:

```
result = session.run("MATCH (knight:Person:Knight) WHERE knight.castle =
{castle} "
                     "RETURN id(knight) AS knight_id", {"castle": "Camelot"})
for record in result:
    session.run("MATCH (knight) WHERE id(knight) = {id} "
                "MATCH (king:Person) WHERE king.name = {king} "
                "CREATE (knight)-[:DEFENDS]->(king)", {"id":
record["knight_id"], "king": "Arthur"})
```

4.3.4.3 结果摘要

Cypher 语句的元数据可以与结果集一起展现出来，方法是通过在 Cypher 语句前面添加 PROFILE 或 EXPLAIN 关键字来获得查询操作的运行计划。

以下示例将通过在数据库中运行查询操作并输出其运行计划。因为在 Cypher 指令前带有“PROFILE”关键字，所以本次查询操作将生成一个运行计划，通过 IresultSummary 对象的 Profile 属性可以获得此运行计划（有关查询配置文件的更多信息，请参见“3.6.2 查询性能分析”小节）。

下面例子是查询并输出执行计划：

C#:

```
var result = session.Run("PROFILE MATCH (p:Person {name: {name}}) RETURN
id(p)",
new Dictionary<string, object> { { "name", "Arthur" } });
IResultSummary summary = result.Consume();
```

```
Output.WriteLine(summary.StatementType.ToString());
Output.WriteLine(summary.Profile.ToString());
```

Java:

```
try ( Transaction tx = session.beginTransaction() )
{
    StatementResult result = tx.run( "PROFILE MATCH (p:Person { name:
{name} }) RETURN id(p)",
            Values.parameters( "name", "Arthur" ) );
    ResultSummary summary = result.consume();
    System.out.println( summary.statementType() );
    System.out.println( summary.profile() );
}
```

JavaScript:

```
session
  .run("PROFILE MATCH (p:Person {name: {name}}) RETURN id(p)", {name:
"Arthur"})
  .then(function (result) {
    //_console.log(result.summary.profile);
    console.log(result.summary.profile);
    session.close();
  });
```

Python:

```
result = session.run("PROFILE MATCH (p:Person {name: {name}}) "
                     "RETURN id(p)", {"name": "Arthur"})
summary = result.consume()
print(summary.statement_type)
print(summary.profile)
```

Neo4j 还可以为查询提供通知，例如，当查询将导致构建笛卡尔乘积的数据量时的警告。以下示例将生成带有估算数据量的解释计划，但不在数据库中实际运行查询。

C#:

```
var summary = session.Run("EXPLAIN MATCH (king), (queen) RETURN king,
queen").Consume();
foreach (var notification in summary.Notifications)
{
    Output.WriteLine(notification.ToString());
}
```

Java:

```
try ( Transaction tx = session.beginTransaction() )
{
    ResultSummary summary = tx.run( "EXPLAIN MATCH (king), (queen) RETURN
king, queen" ).consume();
    for ( Notification notification : summary.notifications() )
    {
        System.out.println( notification );
    }
```

```
}
```

JavaScript:

```
session
  .run("EXPLAIN MATCH (king), (queen) RETURN king, queen")
  .then(function (result) {
    var notifications = result.summary.notifications, i;
    for (i = 0; i < notifications.length; i++) {
      console.log(notifications[i].code);
    }
    session.close();
  });
```

Python:

```
result = session.run("EXPLAIN MATCH (king), (queen) RETURN king, queen")
summary = result.consume()
for notification in summary.notifications:
    print(notification)
```

驱动程序还可以查看查询执行的时间，例如 resultAvailableAfter，它可以返回服务器执行查询所花费的时间。更多详细信息，请参阅 Neo4j 官网 API 查阅手册的驱动程序 API 章节[16]。

4.3.5 数据类型

Neo4j 具有用于处理和存储数据类型的系统。驱动程序可以将数据在应用程序语言类型和 Neo4j 类型系统之间进行转换。为了能够理解传递参数和过程结果的数据类型，需要知道 Neo4j 类型系统的基础知识和理解 Neo4j 类型在驱动程序中的映射关系。

以下 Neo4j 类型对参数和结果都是有效的：

- Boolean
- Integer
- Float
- String
- List
- Map

Map 和 List 可以由数据库处理并在语句中使用。List 的每个元素可以是 Neo4j 的基本类型（如 Integer、Float 等），也可以存储为节点和关系上的属性。节点或关系的属性不能使用 Map 和 List 相组合的类型。

除了对参数和结果有效的类型外，以下 Neo4j 类型仅对结果有效：

- Node
- Relationship
- Path

[16] https://neo4j.com/docs/api/java-driver/current/org/neo4j/driver/v1/summary/ResultSummary.html#resultAvailableAfter-java.util.concurrent.TimeUnit-

Neo4j 驱动程序将 Neo4j 类型映射为本地语言类型，下面是将 Neo4j 类型映射为本地语言类型，如表 4-5~表 4-8 所示。

表 4-5　Neo4j 类型映射为 C#版本

Neo4j	.Net
Null	Null
Boolean	System.Boolean
Integer	System.Int64
Float	System.Double
String	System.String
List	System.CollectionsGeneric.IList<T>
Map	System.Collectons.IDictionary<TKey,TValue>
Node	Neo4j.Driver.V1.Ie (*)
Relationship	Neo4j.Driver.V1.IRelationship (*)
Path	Neo4j.Driver.V1.IPath (*)

表 4-6　Neo4j 类型映射为 Java 版本

Neo4j	Java
Null	Null
Boolean	Java.lang.Boolean
Integer	Java.lang.Long
Float	Java.lang.Double
String	Java.lang.String
List	Java.util.List<T>
Map	Java.util.Map<K,V>
Node	Org.neo4j.driver.V1.types.Node (*)
Relationship	Org.neo4j.driver.V1.types.Relationship (*)
Path	Org.neo4j.driver.V1.Path (*)

表 4-7　Neo4j 类型映射为 Javascript 版本

Neo4j	JavaScript
Null	Null
Boolean	Boolean
Integer	Integer (*)
Float	Number
String	String
List	Array
Map	Object
Node	Node (*)
Relationship	Relationship(*)
Path	Path (*)

表 4-8　Neo4j 类型映射为 Python 版本

Neo4j	.Net
Null	None
Boolean	bool
Integer	Int/long (*)
Float	float

（续表）

Neo4j	.Net
String	Str/Unicode (*)
List	List
Map	dict
Node	Node (*)
Relationship	Relationship(*)
Path	Path (*)

4.3.6 异常

异常列表说明如下。

（1）ClientException

ClientException 指示客户端执行的操作不正确，提供的错误代码可用于确定问题的详细原因。

（2）ConnectionFailureException

ConnectionFailureException 指示底层连接中存在问题，它很可能已被终止。

（3）DatabaseException

DatabaseException 指示底层数据库中存在问题，提供的错误代码可用于确定问题的详细原因。

（4）NoSuchRecordException

每当客户端期望读取不可用的记录（因为它不是由服务器返回的）时抛出，通常表示客户端代码和数据库应用程序逻辑之间的期望不匹配。

（5）ServiceUnavailableException

ServiceUnavailableException 表示驱动程序无法与提供的端点通信，在路由 URI 的情况下，可能需要重新创建驱动程序以解决此错误。

（6）SessionExpiredException

SessionExpiredException 表示会话不再满足获取它的条件。例如，服务器不再接受写入请求。需要从驱动程序获取新会话，并且必须重新执行过期会话上执行的所有操作。

（7）TransientException

TransientException 用信号通知可能通过重试来解决的临时故障，提供的错误代码可用于确定问题的详细原因。

下面是程序中捕获异常的例子：

C#:

```
try
{
    session.Run("This will cause a syntax error").Consume();
}
catch (ClientException)
```

```
{
    throw new InvalidOperationException("Something really bad has
happened!");
}
```

Java:

```
try
{
    try ( Transaction tx = session.beginTransaction() )
    {
        tx.run( "This will cause a syntax error" ).consume();
    }
}
catch ( ClientException e )
{
```

JavaScript:

```
session
  .run("This will cause a syntax error")
  .catch( function(err) {
    console.log(err);
    session.close();
  });
```

Python:

```
try:
    session.run("This will cause a syntax error").consume()
except CypherError:
    raise RuntimeError("Something really bad has happened!")
finally:
    session.close()
```

4.3.7　其他语言驱动

由于 Neo4j 社区的贡献，几乎每一种流行的编程语言都有驱动程序，其中大部分模拟现有的数据库驱动程序习惯和方法。读者可以访问 Neo4j 官方网站下载如表 4-9 所示的驱动程序。

表 4-9　Neo4j 其他语言的驱动

.NET	Java	Spring	Neo4j-OGM	JavaScript
Python	Ruby	PHP	R	Go
Erlang/Elixir	C/C++	Clojure	Perl	Haskell

4.4 Neo4j HTTP API

Neo4j HTTP API 是专门针对跨平台操作开发出来的一套与开发平台、开发语言无关的API，我们可以使用任何编程语言来调用 Neo4j HTTP API。

我们知道 RESTful 是基于 HTTP 协议的、使用 JSON 作为数据格式来传输数据。同样，Neo4j HTTP API 默认情况下传输的也是 JSON，客户端通过 POST / PUT 请求将 JSON 数据发送到 Neo4j HTTP API，Neo4j 完成相应操作后再将结果以 JSON 形式返回到客户端。

4.4.1 流

向 Neo4j HTTP API 发出请求后的响应可以作为 JSON 流传输，从而在服务器端实现更高的性能和更低的内存开销。如果要使用流式处理，我们需要每个 HTTP 请求添加请求头部 X-Stream:true。

为了加快查询，在重复的情况下尽量不要使用文字，而是尽可能用参数替换它们。这将让服务器缓存查询计划。

4.4.2 认证和授权

本节介绍使用 Neo4j HTTP API 的身份验证和授权。

HTTP API 支持身份验证和授权，因此必须使用有效用户的用户名和密码对 HTTP API 的请求授权。默认情况下身份验证和授权是启用的，有关如何启用和禁用身份验证和授权的说明，请参见“5.3.2 认证和授权”小节中的启用身份验证和授权实例，或“附录 A Neo4j 配置设置”。

当 Neo4j 首次安装时，可以使用默认用户 neo4j 和默认密码 neo4j 进行身份验证。在允许访问资源之前，必须更改默认密码。

4.4.2.1 认证

1. 缺少认证

如果未提供授权头，则服务器将返回错误。

请求

```
GET http://localhost:7474/db/data/
Accept: application/json; charset=UTF-8
```

返回结果

```
401: Unauthorized
Content-Type: application/json; charset=UTF-8
WWW-Authenticate: Basic realm="Neo4j"
```

```
{
  "errors" : [ {
    "code" : "Neo.ClientError.Security.Unauthorized",
    "message" : "No authentication header supplied."
  } ]
}
```

2. 通过验证访问服务器

通过使用 HTTP 基本认证向 Neo4j 发送用户名和密码进行认证。请求应包括授权头，其值为 Basic <用户名密码串>，其中“用户名密码串”是 base64 编码的字符串（如下面例子 Authorization 授权头后面的编码串），进行 base64 编码前的格式为：“用户名：密码”。

请求

GET http://localhost:7474/user/neo4j
Accept: application/json; charset=UTF-8
Authorization: Basic bmVvNGo6c2VjcmV0

返回结果

200: OK
Content-Type: application/json; charset=UTF-8

```
{
  "password_change_required" : false,
  "password_change" : "http://localhost:7474/user/neo4j/password",
  "username" : "neo4j"
}
```

3. 授权出错

如果提供的用户名或密码不正确，服务器将返回错误。

请求

POST http://localhost:7474/db/data/
Accept: application/json; charset=UTF-8
Authorization: Basic bmVvNGo6aW5jb3JyZWN0

返回结果

401: Unauthorized
Content-Type: application/json; charset=UTF-8
WWW-Authenticate: Basic realm="Neo4j"

```
{
  "errors" : [ {
    "code" : "Neo.ClientError.Security.Unauthorized",
```

```
    "message" : "Invalid username or password."
  } ]
}
```

4. 修改密码

在某些情况下，像第一次访问 Neo4j 时，用户将需要选择一个新的密码，数据库将发出需要新密码并拒绝访问的响应。

请求

GET http://localhost:7474/db/data/
Accept: application/json; charset=UTF-8
Authorization: Basic bmVvNGo6bmVvNGo=

返回结果

```
403: Forbidden
Content-Type: application/json; charset=UTF-8
```

```
{
  "password_change" : "http://localhost:7474/user/neo4j/password",
  "errors" : [ {
    "code" : "Neo.ClientError.Security.Forbidden",
    "message" : "User is required to change their password."
  } ]
}
```

4.4.2.2 用户状态和密码更改

假设知道当前密码，可以询问服务器的用户状态。

请求

GET http://localhost:7474/user/neo4j
Accept: application/json; charset=UTF-8
Authorization: Basic bmVvNGo6c2VjcmV0

返回结果

```
200: OK
Content-Type: application/json; charset=UTF-8
```

```
{
  "password_change_required" : false,
  "password_change" : "http://localhost:7474/user/neo4j/password",
  "username" : "neo4j"
}
```

1. 用户首次访问时的状态

在首次使用默认密码访问时，用户状态将指示用户密码需要更改。

请求

GET http://localhost:7474/user/neo4j
Accept: application/json; charset=UTF-8
Authorization: Basic bmVvNGo6bmVvNGo=

返回结果

200: OK
Content-Type: application/json; charset=UTF-8

```
{
  "password_change_required" : true,
  "password_change" : "http://localhost:7474/user/neo4j/password",
  "username" : "neo4j"
}
```

2. 修改用户密码

假设知道当前密码，可以要求服务器更改用户密码。可以选择任何喜欢的密码，只要它不同于当前密码。

请求

POST http://localhost:7474/user/neo4j/password
Accept: application/json; charset=UTF-8
Authorization: Basic bmVvNGo6bmVvNGo=
Content-Type: application/json

```
{
  "password" : "secret"
}
```

返回结果

200: OK

3. 禁用身份验证和授权时的访问

当禁用身份验证和授权时，可以在没有授权标头的情况下发送 HTTP API 请求。

4. 将安全配置从一个实例复制到另一个实例

用户名和密码组合对每个 Neo4j 实例都是仅限于本地的。在许多情况下想要使用预配置的身份验证和授权启动 Neo4j 实例。有关如何执行此操作的说明，请参见“5.3.2.1 本机用户

和角色管理”小节。

4.4.3 在一个请求中使用事务

Neo4j 的事务 HTTP API 允许在事务范围内执行一系列 Cypher 语句。事务可以在多个不同的 HTTP 请求中保持打开状态，直到客户端选择提交或回滚。每个 HTTP 请求包括待执行的 Cypher 语句列表，通常情况下可以用事务来提交 Cypher 语句。

服务器对提交的事务有时间限制。如果在超时期限内事务的请求没有回应，则服务器将会回滚此事务。可以在服务器配置中配置超时时长，方法是将 dbms.transaction_timeout 设置为超时前的秒数，默认超时为 60 秒。

当请求失败时，事务将回滚。通过检查事务密钥是否存在可以确定事务是否仍然处于打开状态。

事务不会在 HA（High Availability，高可用）集群的成员之间共享。因此，如果在 HA 集群中使用此端点，则必须确保事务的所有请求都发送到同一个 Neo4j 实例才行。

如果不需要跨多个 HTTP 请求保持使用同一个事务，则可以使用单个 HTTP 请求来创建事务并执行语句。

【程序 4-41】在一个请求中开始并提交事务

请求

POST http://localhost:7474/db/data/transaction/commit
Accept: application/json; charset=UTF-8
Content-Type: application/json

```
{
  "statements" : [ {
    "statement" : "CREATE (n) RETURN id(n)"
  } ]
}
```

返回结果

200: OK
Content-Type: application/json

```
{
  "results" : [ {
    "columns" : [ "id(n)" ],
    "data" : [ {
      "row" : [ 6 ],
      "meta" : [ null ]
    } ]
  } ],
  "errors" : [ ]
}
```

4.4.4　执行多条语句

可以在同一请求中发送多个 Cypher 语句，得到的响应将包含每个语句的结果。

【程序 4-42】执行多条语句

请求

POST http://localhost:7474/db/data/transaction/commit
Accept: application/json; charset=UTF-8
Content-Type: application/json

```
{
  "statements" : [ {
    "statement" : "CREATE (n) RETURN id(n)"
  }, {
    "statement" : "CREATE (n {props}) RETURN n",
    "parameters" : {
      "props" : {
        "name" : "My Node"
      }
    }
  } ]
}
```

返回结果

```
200: OK
Content-Type: application/json
```

```
{
  "results" : [ {
    "columns" : [ "id(n)" ],
    "data" : [ {
      "row" : [ 2 ],
      "meta" : [ null ]
    } ]
  }, {
    "columns" : [ "n" ],
    "data" : [ {
      "row" : [ {
        "name" : "My Node"
      } ],
      "meta" : [ {
        "id" : 3,
        "type" : "node",
        "deleted" : false
      } ]
    } ]
```

```
  } ],
  "errors" : [ ]
}
```

4.4.5 运行一个事务

可以通过将多个 Cypher 语句发送到事务端点来创建新事务，服务器将响应发送语句的结果，以及打开的事务的位置。

请求

```
POST http://localhost:7474/db/data/transaction
Accept: application/json; charset=UTF-8
Content-Type: application/json
```

```
{
  "statements" : [ {
    "statement" : "CREATE (n {props}) RETURN n",
    "parameters" : {
      "props" : {
        "name" : "My Node"
      }
    }
  } ]
}
```

返回结果

```
201: Created
Content-Type: application/json
Location: http://localhost:7474/db/data/transaction/10
```

```
{
  "commit" : "http://localhost:7474/db/data/transaction/10/commit",
  "results" : [ {
    "columns" : [ "n" ],
    "data" : [ {
      "row" : [ {
        "name" : "My Node"
      } ],
      "meta" : [ {
        "id" : 10,
        "type" : "node",
        "deleted" : false
      } ]
    } ]
  } ],
  "transaction" : {
    "expires" : "Thu, 02 Feb 2017 18:48:33 +0000"
  },
```

```
  "errors" : [ ]
}
```

4.4.6　在打开的事务中执行语句

如果创建了一个开放的事务，可以发送多个请求，每个请求都可以执行 Cypher 语句。

请求

```
POST http://localhost:7474/db/data/transaction/12
Accept: application/json; charset=UTF-8
Content-Type: application/json
```

```
{
  "statements" : [ {
    "statement" : "CREATE (n) RETURN n"
  } ]
}
```

返回结果

```
200: OK
Content-Type: application/json
```

```
{
  "commit" : "http://localhost:7474/db/data/transaction/12/commit",
  "results" : [ {
    "columns" : [ "n" ],
    "data" : [ {
      "row" : [ { } ],
      "meta" : [ {
        "id" : 11,
        "type" : "node",
        "deleted" : false
      } ]
    } ]
  } ],
  "transaction" : {
    "expires" : "Thu, 02 Feb 2017 18:48:33 +0000"
  },
  "errors" : [ ]
}
```

4.4.7　重置超时事务

每个事务在一段时间无任何活动后将会自动过期。这可以通过重置事务超时来防止其过期。

可以通过向服务器发送执行空语句的列表保持活动请求来重置超时。此请求将刷新重置事务过期时间，并在响应的“事务”部分中返回事务作为 RFC 1123 格式化的新时间的时间戳。

请求

```
POST http://localhost:7474/db/data/transaction/2
Accept: application/json; charset=UTF-8
Content-Type: application/json
```

```
{
  "statements" : [ ]
}
```

返回结果

```
200: OK
Content-Type: application/json
```

```
{
  "commit" : "http://localhost:7474/db/data/transaction/2/commit",
  "results" : [ ],
  "transaction" : {
    "expires" : "Thu, 02 Feb 2017 18:48:30 +0000"
  },
  "errors" : [ ]
}
```

4.4.8 提交事务

如果有一个开放的事务，可以提交一个请求，或者可以提交其他语句以及提交事务之前的请求。

请求

```
POST http://localhost:7474/db/data/transaction/6/commit
Accept: application/json; charset=UTF-8
Content-Type: application/json
```

```
{
  "statements" : [ {
    "statement" : "CREATE (n) RETURN id(n)"
  } ]
}
```

返回结果

```
200: OK
Content-Type: application/json
```

```
{
  "results" : [ {
    "columns" : [ "id(n)" ],
    "data" : [ {
```



```
      "row" : [ 5 ],
      "meta" : [ null ]
    } ]
  } ],
  "errors" : [ ]
}
```

4.4.9　回滚事务

如果有一个打开的事务，可以发送回滚请求，服务器将回滚事务。尝试在此事务中运行的任何其他语句将立即失败。

请求

```
DELETE http://localhost:7474/db/data/transaction/3
Accept: application/json; charset=UTF-8
```

返回结果

```
200: OK
Content-Type: application/json; charset=UTF-8
```

```
{
  "results" : [ ],
  "errors" : [ ]
}
```

4.4.10　查询统计信息

通过将语句的 includeStats 设置为 true，将返回查询统计信息。

请求

```
POST http://localhost:7474/db/data/transaction/commit
Accept: application/json; charset=UTF-8
Content-Type: application/json
```

```
{
  "statements" : [ {
    "statement" : "CREATE (n) RETURN id(n)",
    "includeStats" : true
  } ]
}
```

返回结果

```
200: OK
Content-Type: application/json
```

```
{
  "results" : [ {
```

```
    "columns" : [ "id(n)" ],
    "data" : [ {
      "row" : [ 4 ],
      "meta" : [ null ]
    } ],
    "stats" : {
      "contains_updates" : true,
      "nodes_created" : 1,
      "nodes_deleted" : 0,
      "properties_set" : 0,
      "relationships_created" : 0,
      "relationship_deleted" : 0,
      "labels_added" : 0,
      "labels_removed" : 0,
      "indexes_added" : 0,
      "indexes_removed" : 0,
      "constraints_added" : 0,
      "constraints_removed" : 0
    }
  } ],
  "errors" : [ ]
}
```

4.4.11 以图格式返回结果

当查询返回的节点和关系的图结构时，可以指定“图”结果数据格式。当想要显示图形结构时，这个功能是很有用的。该格式整理了结果所有列中的所有节点和关系，并且理顺了节点和关系（包括路径）的集合。

请求

```
POST http://localhost:7474/db/data/transaction/commit
Accept: application/json; charset=UTF-8
Content-Type: application/json
```

```
{
  "statements" : [ {
    "statement" : "CREATE ( bike:Bike { weight: 10 } ) CREATE
( frontWheel:Wheel { spokes: 3 } ) CREATE ( backWheel:Wheel { spokes: 32 } )
CREATE p1 = (bike)-[:HAS { position: 1 } ]->(frontWheel) CREATE p2 = (bike)-
[:HAS { position: 2 } ]->(backWheel) RETURN bike, p1, p2",
    "resultDataContents" : [ "row", "graph" ]
  } ]
```

返回结果

```
200: OK
Content-Type: application/json
```

```
{
```

```
"results" : [ {
  "columns" : [ "bike", "p1", "p2" ],
  "data" : [ {
    "row" : [ {
      "weight" : 10
    }, [ {
      "weight" : 10
    }, {
      "position" : 1
    }, {
      "spokes" : 3
    } ], [ {
      "weight" : 10
    }, {
      "position" : 2
    }, {
      "spokes" : 32
    } ] ],
    "meta" : [ {
      "id" : 7,
      "type" : "node",
      "deleted" : false
    }, [ {
      "id" : 7,
      "type" : "node",
      "deleted" : false
    }, {
      "id" : 0,
      "type" : "relationship",
      "deleted" : false
    }, {
      "id" : 8,
      "type" : "node",
      "deleted" : false
    } ], [ {
      "id" : 7,
      "type" : "node",
      "deleted" : false
    }, {
      "id" : 1,
      "type" : "relationship",
      "deleted" : false
    }, {
      "id" : 9,
      "type" : "node",
      "deleted" : false
    } ] ],
    "graph" : {
      "nodes" : [ {
        "id" : "7",
```




```
          "labels" : [ "Bike" ],
          "properties" : {
            "weight" : 10
          }
        }, {
          "id" : "8",
          "labels" : [ "Wheel" ],
          "properties" : {
            "spokes" : 3
          }
        }, {
          "id" : "9",
          "labels" : [ "Wheel" ],
          "properties" : {
            "spokes" : 32
          }
        } ],
        "relationships" : [ {
          "id" : "0",
          "type" : "HAS",
          "startNode" : "7",
          "endNode" : "8",
          "properties" : {
            "position" : 1
          }
        }, {
          "id" : "1",
          "type" : "HAS",
          "startNode" : "7",
          "endNode" : "9",
          "properties" : {
            "position" : 2
          }
        } ]
      }
    } ]
  } ],
  "errors" : [ ]
}
```

4.4.12 错误处理

针对事务端点的任何请求，其结果都将被返回给客户端。因此，当服务器发送 HTTP 状态代码时，服务器并不知道请求是否成功。因此，对事务端点的所有请求都将返回 200 或 201 状态代码，而不管语句是否已成功执行。在返回的响应内容的末尾，服务器将返回执行语句时发生的错误列表。如果此列表为空，则请求已成功完成。

在执行语句时如果发生任何错误，服务器将回滚事务。

这个例子，向服务器发送一个无效的语句来演示错误处理。



请求

```
POST http://localhost:7474/db/data/transaction/11/commit
Accept: application/json; charset=UTF-8
Content-Type: application/json
```

```
{
  "statements" : [ {
    "statement" : "This is not a valid Cypher Statement."
  } ]
}
```

返回结果

```
200: OK
Content-Type: application/json
```

```
{
  "results" : [ ],
  "errors" : [ {
    "code" : "Neo.ClientError.Statement.SyntaxError",
    "message" : "Invalid input 'T': expected <init> (line 1, column 1
(offset: 0))\n\"This is not a valid Cypher Statement.\"\n ^"
  } ]
}
```

4.4.13　在事务中处理错误

当请求中出现错误，服务器将回滚事务。通过检查返回的响应中是否存在事务键，可以判断事务是否仍然打开。

请求

```
POST http://localhost:7474/db/data/transaction/9
Accept: application/json; charset=UTF-8
Content-Type: application/json
```

```
{
  "statements" : [ {
    "statement" : "This is not a valid Cypher Statement."
  } ]
}
```

返回结果

```
200: OK
Content-Type: application/json
```

```
{
  "commit" : "http://localhost:7474/db/data/transaction/9/commit",
```



```
  "results" : [ ],
  "errors" : [ {
    "code" : "Neo.ClientError.Statement.SyntaxError",
    "message" : "Invalid input 'T': expected <init> (line 1, column 1
(offset: 0))\n\"This is not a valid Cypher Statement.\"\n ^"
  } ]
}
```

4.5 其他开发技术介绍

4.5.1 Spring-Data-Neo4j

Spring 对 Neo4j API 进行了封装后生产出了 Spring-Data-Neo4j 库，这对于使用 Spring 或 Spring Boot 框架的 Java 开发人员来说是用来操作 Neo4j 的较好的开发工具，本节将介绍使用 Spring-Data-Neo4j 库访问 Neo4j，包括对象映射、Spring 数据存储库、转换、事务处理等。

在使用 Spring-Data-Neo4j 库之前，我们需要确认自己电脑上装有 JDK 8，因为 Spring-Data-Neo4j 库是在 JDK8 基础上开发出来的。到笔者发稿之日，Spring-Data-Neo4j 最新版本是 Spring-Data-Neo4j 4，它提供了完整的对象图映射功能。Spring-Data-Neo4j 4 基于 Neo4j-OGM 的 Java 对象图映射器，并集成到 Spring 数据基础架构中，包括 Spring-Data 存储库和注解对象映射的支持。

Spring-Data-Neo4j 集成了 Neo4j-OGM 库，提供快速全面的对象图映射。此外，它还提供对 Spring 转换、事务处理、Spring 数据存储库、Spring 数据 REST 和 Spring-Boot 的支持。

下面我们以 Spring-Data-Neo4 的 4.2 版本讨论在项目中的部署，对于后续版本的具体部署方法请读者参阅 Spring Data 官方网站[17]。

4.5.2 Spring-Data-Neo4 在项目中的部署

第一步，我们先添加 Spring-Data-Neo4j 的包文件，我们使用 maven 管理器来添加 Spring-Data-Neo4j 所依赖的包，代码如下：

```
<dependencies>
   <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-neo4j</artifactId>
      <version>{spring-data-neo4j-version}</version>
   </dependency>
</dependencies>
```

[17] https://projects.spring.io/spring-data-neo4j/

第二步，添加 spring 的配置代码：

```
@Configuration
@EnableNeo4jRepositories("org.neo4j.cineasts.repository")
@EnableTransactionManagement
@ComponentScan("org.neo4j.cineasts")
public class PersistenceContext extends Neo4jConfiguration {
    @Override
    public SessionFactory getSessionFactory() {
        return new SessionFactory("org.neo4j.cineasts.domain");
    }
}
```

第三步，创建实体类：

```
@NodeEntity
public class Movie {
  @GraphId Long id;
  String title;
  Person director;
  @Relationship(type="ACTED_IN", direction = Relationship.INCOMING)
  Set<Person> actors = new HashSet<>();
}
```

第四部，声明一个 repository 接口：

```
interface MovieRepository extends GraphRepository<Movie> {
  @Query("MATCH (m:Movie)<-[rating:RATED]-(user)
         WHERE id(m) = {movieId} RETURN rating")
  Iterable<Rating> getRatings(@Param("movieID") Long movieId);
  List<Movie> findByTitle(String title);
}
```

第五步，在需要操作数据的代码处，装配 repository 实例：

```
@Autowired MovieRepository repo;
List<Movie> movie = repo.findByTitle("The Matrix");
Iterable<Rating> ratings = repo.getRatings(movieId);
```

经过以上五步，我们就可以完成 Spring-Data-Neo4j 在我们项目中的部署。

4.5.3　使用 Neo4j-OGM 的对象图映射

开发 Java 业务应用程序通常需要将丰富的域模型映射到数据库。Neo4j-OGM 库是一个纯 Java 库，可以使用 Neo4j 持久化（注解）域对象。它使用 Cypher 语句来处理 Neo4j 中的这些操作。

与 Neo4j 的连接由驱动程序层处理，可以使用二进制协议、HTTP 或 Neo4j 的嵌入式 API。Neo4j-OGM 包如表 4-10 所示。



表 4-10 Neo4j-OGM 包

作者	The Neo4j and GraphAware Teams
包下载地址	http://search.maven.org/#search|ga|1|a%3A%22neo4j-ogm%22
源代码	https://github.com/neo4j/neo4j-ogm
文档	https://neo4j.com/docs/ogm-manual/current/introduction/
文章	https://neo4j.com/blog/neo4j-java-object-graph-mapper-released/
实例	https://github.com/neo4j-examples/neo4j-ogm-university

4.5.4 使用 JDBC 连接 Neo4j

作为一个 Java 开发人员，可能很熟悉 JDBC 作为一种处理关系数据库的方式，直接或通过抽象如 Spring 的 JDBCTemplate 或 MyBatis 来连接到数据库。JDBC 还可以用于 ETL、报表生成器，商业智能。数据管理的许多工具还使用 JDBC 驱动程序与关系数据库进行交互。

由于 Cypher 像 SQL 一样，是一种可以返回表结果的查询语言，因此可以支持 JDBC API 并提供 Neo4j-JDBC 驱动程序。

驱动程序支持 Neo4j 3.x 的新二进制“Bolt”协议、事务 HTTP 端点和 Neo4j 嵌入式连接。

【程序 4-43】使用 JDBC 连接 Neo4j

```
// Connect
Connection con = DriverManager.getConnection("jdbc:neo4j:bolt://localhost");
// Querying
try (Statement stmt = con.createStatement()) {
    ResultSet rs = stmt.executeQuery("MATCH (n:User) RETURN n.name");
    while (rs.next()) {
        System.out.println(rs.getString("n.name"));
    }
}
con.close();
```

有关如何使用它的更多细节可以在我们的 Java-JDBC 示例项目中找到，在那里实现了一个小的 Java webapp，它利用 Neo4j-JDBC 连接到 Neo4j 服务器。

用许多企业工具测试驱动程序，以确保它在各种环境中工作的稳定性。只需下载最新版本的 JAR 并将其添加到工程中就可以使用了。Neo4j-JDBC 包如表 4-11 所示。

表 4-11 Neo4j-JDBC 包

作者	Developers from Larus BA Italy and Neo4j
包下载地址	https://github.com/neo4j-contrib/neo4j-jdbc/releases/latest
源代码	https://github.com/neo4j-contrib/neo4j-jdbc
文档	https://github.com/neo4j-contrib/neo4j-jdbc/blob/master/README.adoc
文章	https://github.com/neo4j-examples/movies-java-jdbc
实例	http://neo4j.com/blog/couchbase-jdbc-integrations-neo4j-3-0/



4.5.5　JCypher

JCypher 以不同的抽象级别为 Neo4j 提供了无缝集成的 Java 访问方式：

在最顶层的抽象层，JCypher 允许将复杂的业务域映射到图数据库。可以获取域对象或 POJO 的任意复杂图形，并将其直接存储到 Neo4j 中。不需要以任何方式修改域对象类，也不需要注解。JCypher 提供了一个开箱即用的默认映射。

JCypher 在抽象的底层，以流畅的 Java API 形式的本地 Java DSL 允许直观和舒适地制定针对 Neo4j 的查询。JCypher 包如表 4-12 所示。

表 4-12　JCypher 包

作者	Wolfgang Schützelhofer
包下载地址	http://search.maven.org/#search|gav|1|g%3A%22net.iot-solutions.graphdb%22%20AND%20a%3A%22jcypher%22
源代码	http://github.com/Wolfgang-Schuetzelhofer/jcypher
文档	http://jcypher.iot-solutions.net/
文章	https://neo4j.com/blog/jcypher-focus-on-your-domain-model-not-how-to-map-it-to-the-database/

4.5.6　Groovy&Grails：Neo4j Grails 插件

GORM 对 Neo4j 提供了一个“尽可能完整的”GORM 实现，将域类和实例映射到 Neo4j 节点空间。Neo4j Grails 如表 4-13 所示，支持以下功能：

- 从 Neo4j 节点编组到 Groovy 类型。
- 支持 GORM 动态查找器、条件和命名查询。
- 使用 Session 会话管理事务。
- 访问 Neo4j 的遍历功能。
- 访问所有类型的 Neo4j 图数据库（嵌入式、REST 和 HA）。

表 4-13　Neo4j Grails 包

作者	Stefan Armbruster, Graeme Rocher
包下载地址	http://www.grails.org/plugin/neo4j
源代码	https://github.com/grails/grails-data-mapping/tree/master/grails-datastore-gorm-neo4j
文档	http://grails.github.io/grails-data-mapping/latest/neo4j/

4.5.7　Clojure：Neocons

Neocons 是一个惯用的、功能丰富的 Clojure 客户端，支持（几乎）所有 Neo4J REST API 功能并不断测试，防止出现边缘服务器更改，如 Cypher 语言改进。Neocons 包如表 4-14 所示。

表 4-14　Neocons 包

作者	Michael Klishin, Rohit Aggarwal
包下载地址	https://clojars.org/clojurewerkz/neocons
源代码	https://github.com/michaelklishin/neocons
文档	http://clojureneo4j.info/

4.5.8 Scala：AnormCypher

一个以 Cypher 为导向的 Scala 库在 Play 后建模框架 Anorm 库，AnormCypher 包如表 4-15 所示。

表 4-15 AnormCypher 包

作者	Eve Freeman
源代码	https://github.com/AnormCypher/AnormCypher
文档	http://anormcypher.org

4.5.9 JPA：Hibernate OGM

Hibernate 对象/网格映射器（OGM）与 Neo4j 支持。Hibernate OGM 包如表 4-16 所示。

表 4-16 Hibernate OGM 包

作者	Davide D'Alto, Gunnar Moelling, Emmanuel Bernard
包下载地址	http://search.maven.org/#search|gav|1|g%3A%22org.hibernate.ogm%22%20AND%20a%3A%22hibernate-ogm-neo4j%22
源代码	https://github.com/hibernate/hibernate-ogm/tree/master/neo4j
文档	http://docs.jboss.org/hibernate/ogm/5.0/reference/en-US/html_single/#ogm-neo4j
实例	https://github.com/TimmyStorms/hibernate-ogm-neo4j-example

第 5 章

Neo4j 数据库管理

本章将介绍与 Neo4j 数据库管理的基本知识和基本操作等相关内容，主要包括：

- 部署
- 监控
- 安全管理
- 运维与优化
- 备份与恢复
- 数据库管理相关工具

5.1 部署

Neo4j 企业版可部署在不同的操作系统环境，包括：Linux、Mac OS X、Windows、Debian、Docker 和 CAPI Flash 等。社区版桌面安装程序仅适用于 Mac OS X 和 Windows 操作系统。社区版和企业版均发布了与平台对应的安装程序或 zip/tar 包。本节中未具体指定版本，则适用于所有版本。部署主要包括：容量规划、单实例或集群安装及安装后的相关处理。

5.1.1 系统需求

本节给出运行一个 Neo4j 数据库实例所需的系统需求清单：

- CPU——对大图（即节点和关系众多）而言，Neo4j 数据库性能通常受限于内存容量或磁盘 I/O 速度，计算性能主要与载入内存中的图数据量相关。最低配置：Intel Core i3 ，推荐配置：Intel Core i7、IBM POWER8。
- 内存——图越大则所需内存越多，但需要正确配置以避免垃圾收集操作的混乱。详情请参见“5.4.1 内存调优”小节。最低配置：2GB，推荐配置：16~32GB 或更多。
- 磁盘——在选择存储设备时，除考虑容量外，磁盘性能是最重要的指标。Neo4j 的工作负载更趋向于随机读，因此，无论是固态硬盘或机械磁盘，优先选择平均寻道时间低的存储介质。详情请参见“5.4.5 磁盘、内存及相关提示”小节。最低配置：

10GB SATA，推荐配置：SSD w/ SATA。

- 文件系统——在 Linux 或 UNIX 的内核实现中通常设有缓冲区高速缓存（Buffer Cache）或页面高速缓存（Page Cache），大多数磁盘 I/O 操作都是通过缓存进行的。当数据写入文件时，内核先将该数据复制到其中一个缓存区中，如果该缓冲区尚未写满，则并不将这些数据排到输出队列，而是等待缓存区写满或者在内核需要重用该缓存区以便存放其他磁盘块数据时，再将该缓冲区数据排至输出队列，然后待其到达队首时，才进行实际的 I/O 操作。这种输出方式被称为延迟写（Delayed Write）。延迟写减少了磁盘读写次数，但降低了文件内容的更新速度，使得想写到文件中的数据在一段时间内并未写入磁盘。当系统发生故障时，这种延迟可能会造成文件更新内容的丢失。为确保 Neo4j 数据库的 ACID 特性，文件系统必须支持刷新功能（即支持：fsync、fdatasync 函数）。详情请参见“5.4.4 Linux 文件系统调优”小节，其中讨论了如何给 Linux 文件系统配置到最优性能。最低配置：ext4（或相似）文件系统，推荐配置：ext4、ZFS。
- 软件——Neo4j 需要一个 Java 虚拟机（Java Virtual Machine，JVM），安装所需的软件如表 5-1 所示。方便起见，Neo4j 社区版安装程序中已经包含了 JVM，而其他发行版，包括 Neo4j 企业版都需要提前预安装 JVM。

表 5-1　Neo4j 安装所需的软件

支持的 JVM	OpenJDK 8、Oracle Java 8 或 IBM Java 8
支持的操作系统	Linux （Ubuntu，Debian）或 Windows Server 2012
支持的 CPU 体系结构	X86 或 OpenPOWER（POWER8）

5.1.2　文件位置

此部分包含数据库系统文件在不同 Neo4j 发行版中的存储位置以及运行 Neo4j 所需要的文件权限。默认情况下，Neo4j 安装后的重要文件及目录可以在表 5-2 和表 5-3 所示的相应路径中找到，需要注意：不同的操作系统对应的位置不完全相同。请注意，Data 目录为 Neo4j 内部专属所有，其文件结构可能在不同版本之间发生变动，如需进一步了解请参阅官方文档。

表 5-2　Neo4j 重要目录位置

	Linux or OS X tarball	Debian/Ubuntu .deb	OS X desktop
Configuration	<neo4j-home>/conf/neo4j.conf	/etc/neo4j/neo4j.conf	${HOME}/Documents/Neo4j/neo4j.conf
Data	<neo4j-home>/data	/var/lib/neo4j/data	${HOME}/Documents/Neo4j
Logs	<neo4j-home>/logs	/var/log/neo4j	${HOME}/Documents/Neo4j/logs
Metrics	<neo4j-home>/metrics	/var/lib/neo4j/metrics	${HOME}/Documents/Neo4j/metrics
Import	<neo4j-home>/import	/var/lib/neo4j/import	${HOME}/Documents/Neo4j/import
Bin	<neo4j-home>/bin	/usr/share/neo4j/bin	(in package)
Lib	<neo4j-home>/lib	/usr/share/neo4j/lib	(in package)
Plugins	<neo4j-home>/plugins	/var/lib/neo4j/plugins	(in package)

表 5-3　Neo4j 重要目录位置（续）

	Windows zip	Windows desktop
Configuration	<neo4j-home>\conf\neo4j.conf	%APPDATA%\Neo4j Community Edition\neo4j.conf
Data	<neo4j-home>\data	%APPDATA%\Neo4j Community Edition
Logs	<neo4j-home>\logs	%APPDATA%\Neo4j Community Edition\logs
Metrics	<neo4j-home>\metrics	%APPDATA%\Neo4j Community Edition\metrics
Import	<neo4j-home>\import	%APPDATA%\Neo4j Community Edition\import
Bin	<neo4j-home>\bin	%ProgramFiles%\Neo4j CE 3.0\bin
Lib	<neo4j-home>\lib	(in package)
Plugins	<neo4j-home>\plugins	%ProgramFiles%\Neo4j CE 3.0\plugins

Log 文件位置，如表 5-4 所示。

表 5-4　Neo4j 数据库 Log 文件位置

文件名	描述
neo4j.log	标准日志，其中写有关于 Neo4j 的一般信息
debug.log	在调试 Neo4j 的问题时有用的信息
http.log	HTTP API 的请求日志
gc.log	JVM 提供的垃圾收集日志记录
query.log	记录超过设定查询时间阈值的查询日志（仅限企业版）
security.log	数据库安全事件日志（仅限企业版）
service-error.log	安装或运行 Windows 服务时遇到的错误日志（仅限 Windows）

1. 配置文件位置

可以使用 dbms.directories.*目录下的相关文件对数据库部分路径进行配置。<neo4j-home>、bin 和 conf 的位置可以使用环境变量进行配置，默认配置文件位置如表 5-5 所示。

表 5-5　Neo4j 数据库配置文件位置

位置	默认值	环境变量	备注
<neo4j-home>	bin 的父目录	NEO4J_HOME	如果 bin 不是子目录，则必须显式设置
bin	Neo4j 脚本所在目录	NEO4J_BIN	如果 neo4j 脚本作为符号链接调用，则必须显式设置
conf	<neo4j-home>/conf	NEO4J_CONF	如果它不是<neo4j-home>的子目录，则必须显式设置

2. 权限设置

Neo4j 数据库运行的用户必须对相应的文件夹及文件具有以下权限：

- 只读权限：conf、import、bin、lib、plugins。
- 读写权限：data、logs、metrics。
- 执行权限：bin 目录中的所有文件。

5.1.3　重要端口

应用程序运行通常都需要占用相应的操作系统端口，如果需要对系统外提供服务，还要

在防火墙等安全设备中开放相关端口的访问，系统部署时须根据特定要求对防火墙进行相应的配置。表 5-6 为 Neo4j 数据库使用到的主要端口。

表 5-6 Neo4j 数据库重要端口

名称	默认端口	相关设置	备注
备份	6362	dbms.backup.enabled=true dbms.backup.address=0.0.0.0:6362	默认情况下禁用备份功能。在生产环境中，对此端口的外部访问由防火墙控制
HTTP 服务	7474		建议在生产环境中不打开此端口用于外部访问，因为流量未加密，由 Neo4j 浏览器使用，也由 REST API 使用
HTTPS 服务	7473		由 REST API 使用
Bolt	7687		由 Cypher Shell 和 Neo4j 浏览器使用
因果集群	5000 6000 7000	causal_clustering.discovery_listen_address=:5000 causal_clustering.transaction_listen_address=:6000 causal_clustering.raft_listen_address=:7000	列出的端口是 neo4j.conf 中的默认端口，端口在实际安装中可能不同，须做相应的修改
高性能集群	5001, 6001	ha.host.coordination=127.0.0.1:5001 ha.host.data=127.0.0.1:6001	列出的端口是 neo4j.conf 中的默认端口，端口在实际安装中可能不同，须做相应的修改
Graphite 监控	2003	metrics.graphite.server=localhost:2003	Neo4j 数据库与 Graphite 服务器通信的端口
JMX 监控	3637	dbms.jvm.additional=-Dcom.sun.management.jmxremote.port=3637	JMX 监控端口。不推荐采用这种检查数据库的方式，默认情况下不启用
Neo4j-shell	1337	dbms.shell.port=1337	neo4j-shell 工具已被弃用，建议停止使用

5.1.4 设置初始密码

使用 neo4j-admin 的 set-initial-password 命令设定当前用户的 neo4j 数据库密码，此操作须在首次启动数据库之前执行。语法为：neo4j-admin set-initial-password <password>，例如：在首次启动数据库之前，将本机 neo4j 用户的密码设置为“h6u4%kr”，相应的命令行操作为：

```
$neo4j-home> bin/neo4j-admin set-initial-password h6u4%kr
```

如果未使用此方法显式设置密码，Neo4j 数据库将其设置为默认密码 neo4j，该密码可在首次登录时依据提示来更改此默认密码。

5.1.5 等待 Neo4j 启动

启动 Neo4j 后，可能需要等待一段时间以便数据库能正常提供服务请求，如果超时，服务依然不可用，则可以重启；或检查是否有网络故障，或是其他原因导致的短暂中断等。可以在 Neo4j 完全启动后，对 Bolt 或 HTTP 端口进行检查，直至数据库能正常响应。需要检查

的事项包括：客户端是使用 HTTP 还是 Bolt 方式连接、是否启用加密或身份验证。设置超时可以防止 Neo4j 启动失败，通常为 10 秒，但数据库在进行恢复或升级时，则可能需要更长时间，具体时间的长短取决于数据库中数据量的多少。如果是集群数据库实例的一部分，则需要等待集群中其他实例都启动并且形成可用的集群。

下面是使用 HTTP 方式在 Bash 中写入的轮询示例，在该示例中禁用了加密和身份验证，示例的命令行中用到了 Curl（CommandLine Uniform Resource Locator，一款利用 URL 语法在命令行方式下工作的开源文件传输工具）[18]，Curl 将对提供的 URL 执行 HTTP GET 操作，并且返回正文文本信息。示例中的%{http_code}为需要测试轮询的 HTTP GET 操作命令，返回值为 200，即表明数据库可正常访问，如果为：404，则表明不存在，数据库不可访问。相关细节可参见“6.2.3.1 状态信息端点介绍”小节。

```
end="$((SECONDS+10))"
while true; do
    [[ "200" = "$(curl --silent --write-out %{http_code} --output /dev/null
http://localhost:7474)" ]] && break
    [[ "${SECONDS}" -ge "${end}" ]] && exit 1
    sleep 1
done
```

5.1.6　使用数据收集器

Neo4j 使用数据收集器（Usage Data Collector，UDC）收集使用数据的信息，并提供给官方的 UDC 服务器（UDC-Server，其网址为：udc.neo4j.org），该数据收集器可以禁用，并且不收集任何机密数据。Neo4j 研发团队使用这些信息作为 Neo4j 社区自动反馈的一种形式，希望通过下载统计信息与使用情况来进行匹配验证。如果某版本收集的数据更多，则其版本的更改会更快。Neo4j 团队非常关注客户隐私，不披露任何个人身份信息。UDC 将收集如下信息：

- 内核版本：Neo4j 内部版本号。
- 商店 ID：在创建数据库的同时创建的随机化全局唯一 ID。
- Ping 计数：UDC 保存一个内部计数器，该内部计数器对于每次 ping 都会递增，并在内核每次重新启动时重置。
- 资料来源：为 neo4j 或 maven。如果从 Neo4j 官网网站下载则资料来源为 neo4j，而在 Maven 网站获得的 Neo4j 版本，则资料来源为 maven。
- Java 版本：引用字符串显示正在使用的 Java 版本。
- 注册 ID：用于注册的服务器实例。
- 关于执行上下文的标签（例如：测试、语言、Web 容器、应用程序容器、spring、ejb 等）。
- Neo4j 版本（社区版或企业版）。
- 当前集群名称的哈希值（如果部署为集群的话）。

[18] https://curl.haxx.se/

- Linux 的发布信息（rpm、dpkg、unknown）。
- User-Agent 标头，用于跟踪 REST 客户端驱动程序的使用情况。
- MAC 地址，以唯一标识配置有防火墙的数据库实例。
- 处理器数量。
- 内存容量。
- JVM 堆大小。
- 数据库节点数、关系数、标签数和属性数。

Neo4j 数据库默认配置是开启了 UDC 程序，并伴随数据库启动而自动运行，UDC 将在数据库正常运行的 10 分钟后才发送第一个 ping 命令，这样做基于两个方面的考虑：首先，不希望启动 UDC 而使数据库系统变得更慢，其次，希望将自动 ping 测试保持最少，而使用 HTTP GET 方式与 UDC 服务器进行 ping 操作。UDC 可以在数据库配置中禁用，在数据库安装目录中的 neo4j.conf 文件，将参数 dbms.udc.enabled 设置为 false，可以轻松地关闭 UDC，该参数的默认值为 true，即默认为开启。neo4j.conf 配置详见“附录 A Neo4j 配置设置”。

5.1.7 配置 Neo4j 连接器

应用程序端与 Neo4j 数据库之间通信需要有相应的机制作保障，从而 Neo4j 连接器（Neo4j connector）应运而生，Neo4j 提供了非常丰富的连接机制，支持 Bolt 二进制协议或 HTTP/HTTPS 方式，极大地方便了应用程序的开发。默认情况下可配置三种不同的 Neo4j 连接器：Bolt 连接器、HTTP 连接器和 HTTPS 连接器，如表 5-7 所示。配置 HTTPS 连接器请参见“5.1.8 安装证书”小节。

表 5-7 Neo4j 连接器配置

连接名	协议	默认端口
dbms.connector.bolt	Bolt 二进制协议	7687
dbms.connector.http	HTTP 协议	7474
dbms.connector.https	HTTPS 协议	7473

每种 Neo4j 连接器均有三个参数：enabled、listen_address、advertised_address，参数的配置信息及说明如表 5-8 所示。

表 5-8 Neo4j 连接器参数配置

参数名	默认值	说明
enabled	true	允许启用或禁用客户端连接器
listen_address	IP 地址:连接器端口	输入连接的 IP 地址
advertised_address	IP 地址:连接器端口	客户端使用此连接器的 IP 地址

- enabled 参数——启用或禁用连接器。禁用时，Neo4j 数据库不再侦听相应端口上的所有传入连接。例如，dbms.connector.https.enabled = false，表明禁用 HTTPS 连接器。注意：不能禁用 HTTP 连接器。要阻止客户端通过 HTTP 连接到数据库，可使用防火墙来阻止 HTTP 端口，或将 HTTP 连接器的 listen_address 配置为仅侦听环回接口（即本机 IP 地址或 localhost），从而阻止与远程客户端的连接。

- listen_address 参数——设置指定 Neo4j 如何侦听传入连接，它由两部分组成：网络 IP 地址（例如 localhost 或 0.0.0.0）和端口号（例如 7687），并以格式“网络 IP 地址：端口号”表示。例如设置 dbms.connector.bolt.listen_address=0.0.0.0:7000，表明要侦听所有网络接口（0.0.0.0）和端口 7000 上的 Bolt 连接。
- advertised_address 参数——设置指定客户端使用该连接器的地址。在因果集群中很有用，它允许每个服务器可以正确地广播集群中其他服务器的地址。公告的地址由两部分组成：网络 IP 地址（例如 localhost 或 0.0.0.0）和端口号（例如 7687），并以格式“网络 IP 地址：端口号”表示。如果使用代理路由或端口映射，则可以单独为每个连接器指定 advertised_address。例如，如果 Neo4j 服务器上的端口 7687 是从外网的 9000 端口映射过来的，则将 bolt 连接器指定 advertised_address 参数为：dbms.connector.bolt.advertised_address=<server-name>:9000。

dbms.connectors.default_listen_address 和 dbms.connectors.default_advertised_address 参数可用于分别指定 listen_address 和 advertised_address 的网络接口部分，如无特别配置，可为所有连接器设置默认值。这两个参数的说明如表 5-9 所示。

表 5-9　Neo4j 连接器监听地址参数配置

参数名	默认值	说明
dbms.connectors.default_listen_address	localhost	所有连接器的 listen_address 的默认网络接口地址
dbms.connectors.default_advertised_address	localhost	所有连接器的 advertised_address 的默认网络接口地址

- default_listen_address 参数——监听地址由两部分组成：网络接口地址（例如 localhost 或 0.0.0.0）和端口号（例如 7687）。如果未指定 listen_address 的网络接口地址，将继承共享设置 default_listen_address。例如：要侦听所有网络接口（0.0.0.0）和端口 7000 上的 Bolt 连接，可将 Bolt 连接器设置 listen_address 参数为：dbms.connector.bolt.listen_address=0.0.0.0:7000。此方法等价于先设置参数 default_listen_address，再指定 Bolt 连接器的端口号。

```
dbms.connectors.default_listen_address=0.0.0.0
dbms.connector.bolt.listen_address=:7000
```

- default_advertised_address 参数——广播地址由两部分组成：网络接口地址（主机名或 IP 地址）和端口号（例如 7687）。如果未指定 advertised_address 的网络接口地址，则继承参数 default_advertised_address 的设置。例如：指定客户端访问的 Bolt 连接器地址为：dbms.connector.bolt.advertised_address=server1:9000。等价于用 default_advertised_address 设置网络接口，再指定 Bolt 连接器的端口号，即：

```
dbms.connectors.default_advertised_address=server1
dbms.connector.bolt.advertised_address=:9000
```

5.1.8　安装证书

默认情况下，Neo4j 数据库在与官方驱动程序配套使用时，将采用 TLS（Transport Layer

Security，传输层安全）协议加密所有客户服务器通信，包括 Bolt 和 HTTP 协议，这能确保应用程序与数据库之间的安全可靠通信。而客户端如果是浏览器，要与 Neo4j 数据库进行通信，尤其是在生产系统中，则同样要考虑受信任的加密传输，比如：HTTPS 方式，这样可考虑安装 SSL（Secure Socket Layer，安全套接字层）证书，此类证书必须由受信任的证书颁发机构（Certification Authority，CA）颁发，比如：收费 SSL 证书有 VeriSign，免费的有 StartSSL 等。SSL 证书由<file-name>.key 文件和<file-name>.cert 文件组成。Neo4j 要求这两个文件必须分别命名为 neo4j.key 和 neo4j.cert。其中，密钥文件 neo4j.key 是不加密的，需正确设置该文件的权限，以便只有 Neo4j 用户能够读取它。先将此文件放入指定的目录，默认在<neo4j-home>\certificates 目录下，也可通过在 neo4j.conf 中设置 dbms.directories.certificates 来指定证书文件的存放路径，此配置参数有效值为：文件系统路径，一个相对于安装根路径<neo4j-home>的相对路径，默认值为：certificates。neo4j.conf 配置详见“附录 A Neo4j 配置设置”。如果要启用自己的证书，其步骤如下：

（1）关闭 Neo4j 数据库。

（2）获取由受信任的证书颁发机构颁发的 SSL 证书，将对应的文件名重命名为：neo4j.key 和 neo4j.cert。

（3）将这两个文件覆盖 Neo4j 数据库原来的 SSL 证书同名文件，默认情况下，在<neo4j-home>\certificates 目录下，或者为 neo4j.conf 配置参数 dbms.directories.certificates 所设定的路径。

（4）重启 Neo4j 数据库。

Neo4j 支持链接的 SSL 证书，所有证书都必须是 PEM（Privacy Enhanced Mail，隐私增强邮件）格式，并且必须合并到一个文件中，私钥也需要采用 PEM 格式，可支持多主机和通配符证书。如果 Neo4j 已配置多个连接器绑定到不同的接口，则可以使用这种方式的证书。

如果在未安装任何证书的情况下启动，Neo4j 进程将自动生成自签名 SSL 证书和私钥。由于此证书是自签名的，因此不够安全，显然不适合于生产系统。如果在 Neo4j 配置了多个绑定到不同 IP 地址的连接器时，则自签名 SSL 证书方式将不再发挥作用，如需要使用多个 IP 地址，只能手动配置证书，并使用多主机或通配符证书方式。

5.2 监控

本章介绍如何利用 Neo4j 监控工具来记录和显示服务器的各项指标，可通过 Neo4j 输出的度量及查询指标连续分析，也可以采集日志用于连续分析或特定研究。Neo4j 数据库可记录安全事件日志和查询日志，提供查询管理功能，还支持因果集群监控的点对点分析（Ad-hoc Analysis）。Neo4j 数据库同时提供了一个简单直观的 Web 监控界面，通过浏览器方式访问 Neo4j 数据库（默认为：http://localhost:7474/，如果修改了 IP 地址，则需将 localhost 替换为安装 Neo4j 数据库的 IP 地址；如果配置 HTTP 端口为其他端口，则需将 7474 改为设定的

端口），将呈现如图 5-1 所示的访问界面。

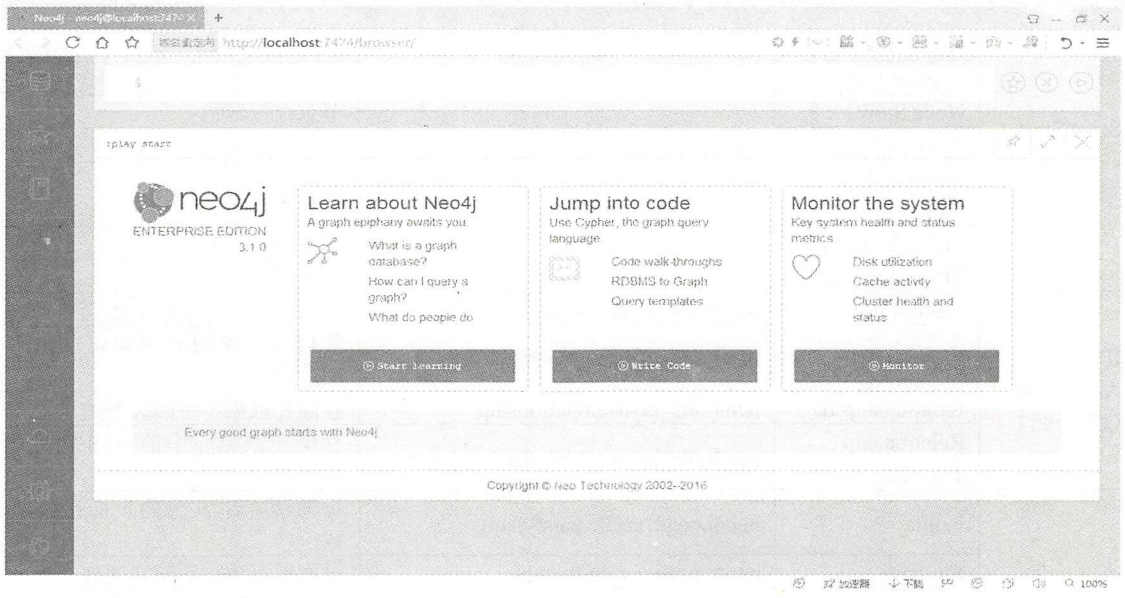


图 5-1　Neo4j 浏览器方式访问界面

其中第三项（Monitor the system，将实时监控 Neo4j 数据库的运行状况，包括：系统健康状态和重要的指标，比如：磁盘利用情况、缓存活跃情况、集群情况等），单击上图的“Monitor”按钮，将进入如图 5-2 所示的监控界面。

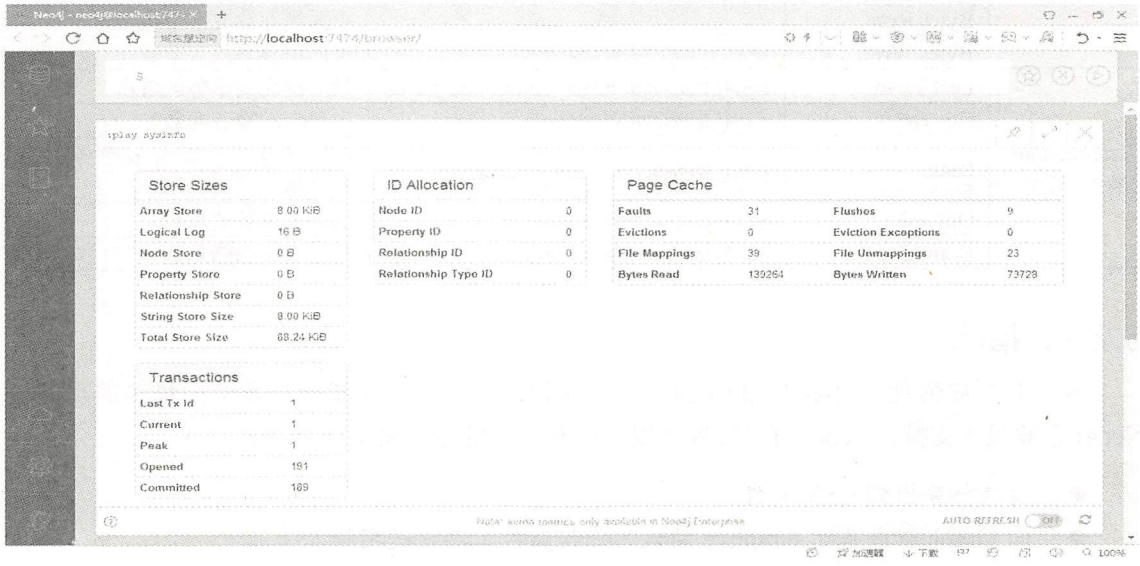


图 5-2　Neo4j 浏览器方式监控界面

其中参数的对应关系如表 5-10 所示，并附上每项参数的简要描述，在“5.2.1 指标”小节中将对 Neo4j 数据库所有监控参数做详细的介绍。

表 5-10 Neo4j 监控界面参数对应关系

类别	英文名称	对应指标	描述
Store Sizes（存储容量）	Array Store	调用操作系统获取 Neo4j 数据库存储文件的大小	数组存储容量
	Logical Log		逻辑日志容量
	Node Store		节点存储容量
	Property Store		属性存储容量
	Relationship Store		关系存储容量
	String Store Size		字符串存储容量
	Total Store Size		总存储容量
ID Allocation（ID 分配）	Node ID	neo4j.ids_in_use.node	存储在数据库中的节点总数
	Property ID	neo4j.ids_in_use.property	数据库中使用的不同属性名称的总数
	Relationship ID	neo4j.ids_in_use.relationship	存储在数据库中的关系的总数
	Relationship Type ID	neo4j.ids_in_use.relationship_type	存储在数据库中的不同关系类型的总数
Page Cache（页面缓存）	Faults	neo4j.page_cache.page_faults	页面缓存中发生的页面错误总数
	Flushes	neo4j.page_cache.flushes	页面缓存执行的刷新次数
	Evictions	neo4j.page_cache.evictions	由页面缓存执行的页面被替换的次数
	Eviction Exceptions	neo4j.page_cache.eviction_exceptions	在页面缓存被替换过程期间的异常总数
	File Mappings	neo4j.page_cache.pins	页面缓存执行的页面锁定次数
	File Unmappings	neo4j.page_cache.unpins	页面缓存执行的页面未锁定次数
	Bytes Read		页面缓存读取字节数
	Bytes Written		页面缓存写入字节数
Transactions 事务	Last Tx Id	neo4j.transaction.last_committed_tx_id	最后提交的事务的 ID
	Current	neo4j.transaction.active	当前活动事务的数量
	Peak	neo4j.transaction.peak_concurrent	此机器上发生的并发事务的最高峰值
	Opened	neo4j.transaction.started	启动的事务的总数
	Committed	neo4j.transaction.committed	已提交事务的总数

5.2.1 指标

本节介绍如何使用 Neo4j 指标输出工具来记录和显示各种指标。本节中描述的功能仅 Neo4j 企业版才支持。Neo4j 可以配置为以下两种不同的方式来呈现指标：

- 将指标导出到 CSV 文件。
- 向 Graphite 或基于 Graphite 协议的任何监控工具发送指标。

5.2.1.1 启用指标记录

Neo4j 可以输出数据库以下指标：

```
# Setting for enabling all supported metrics.
```

仅供非商业用途或交流学习使用


```
metrics.enabled=true
# Setting for enabling all Neo4j specific metrics.
metrics.neo4j.enabled=true
# Setting for exposing metrics about transactions; number of transactions
started, committed, etc.
metrics.neo4j.tx.enabled=true
# Setting for exposing metrics about the Neo4j page cache; page faults,
evictions, flushes and exceptions, etc.
metrics.neo4j.pagecache.enabled=true
# Setting for exposing metrics about approximately entities are in the
database; nodes, relationships, properties, etc.
metrics.neo4j.counts.enabled=true
# Setting for exposing metrics about the network usage of the HA cluster
component.
metrics.neo4j.network.enabled=true
```

5.2.1.2　Graphite

将以下设置添加到 neo4j.conf 配置文件中，以开启与 Graphite 的集成，从而启动 Neo4j 连接到 Graphite 来监控 Neo4j 的各项指标。

```
# Enable the Graphite integration. Default is 'false'.
metrics.graphite.enabled=true
# The IP and port of the Graphite server on the format <hostname or IP
address>:<port number>.
# The default port number for Graphite is 2003.
metrics.graphite.server=localhost:2003
# How often to send data. Default is 3 minutes.
metrics.graphite.interval=3m
# Prefix for Neo4j metrics on Graphite server.
metrics.prefix=Neo4j_1
```

5.2.1.3　CSV 文件

将以下设置添加到 neo4j.conf 中，以便将各项指标导出到本地的 CSV 文件：

```
# Enable the CSV exporter. Default is 'false'.
metrics.csv.enabled=true
# Directory path for output files.
# Default is a "metrics" directory under NEO4J_HOME.
#metrics.csv.path='/local/file/system/path'
# How often to store data. Default is 3 minutes.
metrics.csv.interval=3m
```



CSV 导出器不会自动定期输出文件，如需此功能，建议配置一个作业以定期归档文件。

5.2.1.4　可用指标

通用指标如表 5-11~表 5-19 所示。



仅供非商业用途或交流学习使用

表 5-11　数据库检查点指标

名称	描述
neo4j.check_point.events	到目前为止执行的检查点事件的总数
neo4j.check_point.total_time	到目前为止检查点所花费的总时间
neo4j.check_point.check_point_duration	检查点事件的持续时间

表 5-12　数据库数据指标

名称	描述
neo4j.ids_in_use.relationship_type	存储在数据库中的不同关系类型的总数
neo4j.ids_in_use.property	数据库中使用的不同属性名称的总数
neo4j.ids_in_use.relationship	存储在数据库中的关系的总数
neo4j.ids_in_use.node	存储在数据库中的节点总数

表 5-13　数据库 PageCache 指标

名称	描述
neo4j.page_cache.eviction_exceptions	在页面缓存被替换过程期间的异常总数
neo4j.page_cache.flushes	页面缓存执行的刷新次数
neo4j.page_cache.unpins	页面缓存执行的页面未锁定次数
neo4j.page_cache.pins	页面缓存执行的页面锁定次数
neo4j.page_cache.evictions	由页面缓存执行的页面被替换的次数
neo4j.page_cache.page_faults	页面缓存中发生的页面错误总数

表 5-14　数据库事务指标

名称	描述
neo4j.transaction.started	启动的事务的总数
neo4j.transaction.peak_concurrent	此机器上发生的并发事务的最高峰值
neo4j.transaction.active	当前活动事务的数量
neo4j.transaction.active_read	当前活动的读事务的数量
neo4j.transaction.active_write	当前活动的写事务的数量
neo4j.transaction.committed	已提交事务的总数
neo4j.transaction.committed_read	已提交读事务的总数
neo4j.transaction.committed_write	已提交写事务的总数
neo4j.transaction.rollbacks	回滚事务的总数
neo4j.transaction.rollbacks_read	回滚读事务的总数
neo4j.transaction.rollbacks_write	回滚写事务的总数
neo4j.transaction.terminated	已终止事务的总数
neo4j.transaction.terminated_read	已终止读事务的总数
neo4j.transaction.terminated_write	已终止写事务的总数
neo4j.transaction.last_committed_tx_id	最后提交的事务的 ID
neo4j.transaction.last_closed_tx_id	上次关闭的事务的 ID

表 5-15　Cypher 指标

名称	描述
neo4j.cypher.replan_events	Cypher 决定重新计划一次查询的总次数



仅供非商业用途或交流学习使用

表 5-16　数据库 LogRotation 指标

名称	描述
neo4j.log_rotation.events	到目前为止执行的事务日志的总数
neo4j.log_rotation.total_time	到目前为止日志花费的总时间
neo4j.log_rotation.log_rotation_duration	日志事件的持续时间

表 5-17　网络指标表

名称	描述
neo4j.network.slave_network_tx_writes	从设备到主设备传输数据的字节数
neo4j.network.master_network_store_writes	从一台机器到另外一台机器拷贝传输的字节数
neo4j.network.master_network_tx_writes	主机到从机传输的字节数

表 5-18　集群指标

名称	描述
neo4j.cluster.slave_pull_updates	此实例执行的更新拉取总数
neo4j.cluster.slave_pull_update_up_to_tx	在此实例的最后一次拉更新中拉出的最高事务 ID
neo4j.cluster.is_master	此实例是否是集群中的主节点
neo4j.cluster.is_available	此实例在集群中是否可用

表 5-19　核心指标

名称	描述
neo4j.causal_clustering.core.append_index	追加 RAFT 日志的索引
neo4j.causal_clustering.core.commit_index	RAFT 日志的提交索引
neo4j.causal_clustering.core.term	RAFT 此服务器的术语
neo4j.causal_clustering.core.leader_not_found	尝试提交事务时未找到 Leader
neo4j.causal_clustering.core.tx_pull_requests_received	从读取副本接收的 TX 拉取请求
neo4j.causal_clustering.core.tx_retries	事务重试
neo4j.causal_clustering.core.is_leader	这个服务器是否为领导者
neo4j.causal_clustering.core.dropped_messages	被丢弃的 RAFT 消息数
neo4j.causal_clustering.core.queue_sizes	RAFT 消息排队数

Java 虚拟机指标：Java 虚拟机指标与软硬件环境相关的，主要有硬件配置和 JVM 配置等。通常，这些指标将显示有关垃圾收集的信息（例如事件数和收集时间）、内存池和缓冲区，以及最后运行的活动线程数。

因果集群专用指标：核心服务器（Core Server）和只读副本（Read Replica）两者的特性及支持的协议不同，从而具有不同度量指标。核心服务器指标需监视重要细节，如 Raft 分布式一致性协议的集合状态和已发送到只读副本的事务数。而只读副本指标简单得多，只需跟踪相对于核心服务器的异步复制状态。

核心服务器需记录 Raft 分布式一致性算法相关的各种指标，还需记录其到只读副本（以及任何新提供的在线核心服务器）的负载（以事务日志传送请求），如表 5-20 所示。

表 5-20　核心服务器指标

名称	描述
neo4j.causal_clustering.core.commit_index	显示已安全提交到其 raft 日志中的事务数
neo4j.causal_clustering.core.append_index	显示已经附加（但不一定提交）到其 raft 日志中的事务数
neo4j.causal_clustering.core.term	显示已发生的领导者选举数量的视图



仅供非商业用途或交流学习使用

（续表）

名称	描述
neo4j.causal_clustering.core.leader_not_found	显示找不到 Raft 协议的领导者的次数
neo4j.causal_clustering.core.tx_pull_requests_received	显示已接收的事务日志传送请求的数量
neo4j.causal_clustering.core.tx_retries	显示必须重试的事务数
neo4j.causal_clustering.core.is_leader	显示是否扮演 Raft 领导者角色
neo4j.causal_clustering.core.dropped_messages	显示由当前服务器拥有与其他核心服务器的通信故障而丢弃的消息数
neo4j.causal_clustering.core.queue_sizes	显示超出到其他核心服务器的 Raft 消息的聚合队列大小

只读副本指标记录相对于核心服务器的复制窗口，如表 5-21 所示。

表 5-21 只读副本指标

名称	描述
pull_updates	异步事务更新的请求数
pull_update_highest_tx_id_requested	从核心服务器接收的最高事务 ID
pull_update_highest_tx_id_received	从核心服务器收到的最后一个事务 ID

5.2.2 日志

Neo4j 日志分为安全日志和查询日志两种，用于记录数据库的查询和发生的安全事件。日志功能仅在 Neo4j 企业版才支持。

5.2.2.1 查询日志

要记录数据库所执行的查询，须将 dbms.logs.query.enabled 参数设置为 true。参数 dbms.logs.query.threshold 用于设定记录查询的阈值，即如果查询执行所花费的时间大于此阈值，则记录此查询，dbms.logs.query.threshold 设置为 0，则记录所有查询。查询日志的文件名为 query.log，存储在 Logs 目录中。可以在 neo4j.conf 配置文件中配置查询日志的轮换，相关参数如表 5-22 所示。neo4j.conf 配置详见“附录 A Neo4j 配置设置”。

表 5-22 查询日志参数

参数名称	默认值	描述
dbms.logs.query.enabled	false	是否记录查询耗时超过配置的阈值 dbms.logs.query.threshold
dbms.logs.query.parameter_logging_enabled	True	是否设定查询耗时超过配置阈值
dbms.logs.query.rotation.keep_number	7	设置保存历史查询日志文件的数量
dbms.logs.query.rotation.size	20MB	查询日志自动轮换的文件大小
dbms.logs.query.threshold	0	如果查询执行所花费的时间长于此阈值，则记录该查询（前提是要启用了查询日志记录）

查询日志文件中记录的内容类似如下：

```
2016-10-27 14:31 ... INFO  0 ms: bolt-session  bolt  johndoe  neo4j-
javascript/0.0.0-dev
client/127.0.0.1:59167
...
2016-10-27 14:32 ... INFO  3 ms: server-session  http  127.0.0.1
```



仅供非商业用途或交流学习使用


```
/db/data/cypher  neo4j - CALL dbms.procedures() - {}
2016-10-27 14:32 ... INFO  1 ms: server-session  http  127.0.0.1
/db/data/cypher  neo4j - CALL dbms.security.showCurrentUs...
2016-10-27 14:32 ... INFO  0 ms: bolt-session  bolt  johndoe  neo4j-
javascript/0.0.0-dev client/127.0.0.1:59167
...
```

5.2.2.2　安全事件日志

Neo4j 数据库安全事件日志功能用于记录所有的安全事件，对于本机用户管理，将记录以下操作：

- 登录尝试，每个默认值都记录成功和不成功的登录。
- 更改用户的密码、管理员和用户。
- 创建和删除用户，包括失败的尝试。
- 创建和删除自定义角色，包括失败的尝试。
- 为用户分配和删除角色，包括失败的尝试。
- 暂停和激活用户。
- 非管理员用户尝试列出用户和角色的尝试失败。

安全事件日志文件名为 security.log，存放在 Logs 目录中，可以在 neo4j.conf 配置文件中配置安全事件日志的轮换。相关的参数如表 5-23 所示。

表 5-23　安全事件日志参数

参数名称	默认值	描述
dbms.logs.security.rotation.size	20MB	安全事件日志自动轮换的文件大小
dbms.logs.security.rotation.delay	300s	设置在最后一次日志轮换发生后，日志可能再次轮换之前的最小时间间隔
dbms.logs.security.rotation.keep_number	7	设置保存历史安全事件日志文件的数量

如果使用 LDAP（Lightweight Directory Access Protocol，轻量目录访问协议）作为身份验证方法，还会记录一些 LDAP 配置错误的情况，以及 LDAP 服务器通信事件和故障。在“5.3.3 与 LDAP 集成”小节中将重点介绍 Neo4j 与 LDAP 的集成。如果需要用到大量的编程交互，例如 REST 方式，建议禁用成功登录的日志记录，即在 neo4j.conf 文件中设置：dbms.security.log_successful_authentication = false。安全事件日志文件中记录的内容类似如下：

```
2016-10-27 13:45:00.796+0000 INFO  [AsyncLog @ 2016-10-27 ...]  [johnsmith]:
logged in
2016-10-27 13:47:53.443+0000 ERROR [AsyncLog @ 2016-10-27 ...]  [johndoe]:
failed to log in: invalid principal or credentials
2016-10-27 13:48:28.566+0000 INFO  [AsyncLog @ 2016-10-27 ...]  [johnsmith]:
created user `janedoe`
2016-10-27 13:48:32.753+0000 ERROR [AsyncLog @ 2016-10-27 ...]  [johnsmith]:
tried to create user `janedoe`: The specified user ...
2016-10-27 13:49:11.880+0000 INFO  [AsyncLog @ 2016-10-27 ...]  [johnsmith]:
added role `admin` to user `janedoe`
```



仅供非商业用途或交流学习使用


```
2016-10-27 13:49:34.979+0000 INFO  [AsyncLog @ 2016-10-27 ...]  [johnsmith]:
deleted user `janedoe`
2016-10-27 13:49:37.053+0000 ERROR [AsyncLog @ 2016-10-27 ...]  [johnsmith]:
tried to delete user `janedoe`: User 'janedoe' does ...
2016-10-27 14:00:02.050+0000 INFO  [AsyncLog @ 2016-10-27 ...]  [johnsmith]:
created role `operator`
```

5.2.3 查询管理

Neo4j 提供了相应的手段可从安全或性能角度对查询语句进行检查。查询日志可用于数据库的连续监控和故障排除，事务超时功能是一种安全措施，操作员可以利用它为查询设定最大的运行时间。查询管理可查看数据库中运行的查询，必要时可终止某个查询。

5.2.3.1 事务超时

执行保护功能可以终止其执行时间超过配置超时时间的事务。要启用执行保护，须将 dbms.transaction.timeout 设置为一个时间值，比如：dbms.transaction.timeout=10s，表示设定事务超时时间为 10 秒。如果将 dbms.transaction.timeout 设置为 0（0 是该参数的默认值）则表示：禁用执行保护。此功能对使用自定义超时（通过 Java API）执行的事务不产生影响，因为自定义超时将覆盖 dbms.transaction.timeout 设置的值。

5.2.3.2 查询管理的步骤

本节介绍如何查看和终止当前正在执行的查询，仅在 Neo4j 企业版中才支持此功能。除非另有说明外，本节已提供了步骤的所有参数。学习本节之前，先了解一下这些术语：

- 管理员（Administrator）：已分配了管理员角色的用户。
- 当前用户（Current User）：当前登录的用户，可调用本章所述的命令。
- 用户（User）：由用户名和凭证组成，凭证用于验证用户身份，比如密码等。用户也可以表示人或某个应用程序等。

1. 列出所有运行的查询

管理员能够查看当前实例中运行的所有查询。当前用户可以查看自己当前正在运行的所有查询。语法为：CALL dbms.listQueries()，其返回值如表 5-24 所示。

表 5-24 列出所有运行查询的返回值

名称	类型	描述
queryId	String	查询 ID
username	String	执行该查询的用户名
query	String	查询本身
parameters	Map	该查询的参数列表
startTime	String	该查询的开始时间
elapsedTime	String	查询已消耗的时间
connectionDetails	String	该查询的连接详细信息
metaData	Map	该事务所关联的元数据



仅供非商业用途或交流学习使用

以下示例显示用户“alwood”当前正在运行两个查询：dbms.listQueries()过程和一个创建大量节点的查询。简洁起见，有些数据已作截断处理。

```
CALL dbms.listQueries()
```

结果

```
+---------+---------+------------------+------+------+---------+----------------+------+
|queryId  |username |query             |parame|start-|elapsedTime|connectionDetails|meta- |
|         |         |                  |ters  |Time  |         |                |Data  |
+---------+---------+------------------+------+------+---------+----------------+------+
|"query-271"|"alwood"|"... dbms.listQueries()"|(empty)|"2016..."|"00:00:..."|"server-sess ..."|(empty)|
|"query-272"|"alwood"|"WITH range(1, 3000...)"|(empty)|"2016..."|"00:00:..."|"server-sess ..."|(empty)|
+---------+---------+------------------+------+------+---------+----------------+------+
2 rows
```

2. 终止多个查询

管理员能终止给定查询 ID 列表的任意组事务。当前用户可以终止其自己创建的任意组事务。语法为：CALL dbms.killQueries(ids)，其中 ids 参数为 List<String>类型，是需要终止的事务 ID 的列表。其返回值如表 5-25 所示。

表 5-25　终止多个查询的返回值

名称	类型	描述
queryId	String	查询 ID
username	String	启动运行该查询的用户名

以下示例显示，管理员已终止分别由用户的 joesmith 和 annebrown 启动的 ID 为'query-378'和'query-765'的查询。

```
CALL dbms.killQueries(['query-378','query-765'])
```

结果

```
+-------------+-------------+
| queryId     | username    |
+-------------+-------------+
| "query-378" | "joesmith"  |
| "query-765" | "annebrown" |
+-------------+-------------+
2 rows
```

3. 终止一个查询

管理员能终止给定查询 ID 的任意一个事务。当前用户可以终止其自己创建的任意一个事务。语法为：CALL dbms.killQuery(id)，其中 id 参数为 String 类型，是需要终止的事务 ID。其返回值如表 5-26 所示。

表 5-26　终止一个查询的返回值

名称	类型	描述
queryId	String	查询 ID
username	String	启动运行该查询的用户名



仅供非商业用途或交流学习使用

以下示例显示用户“joesmith”终止了事务 ID 为“query-502”的查询。

```
CALL dbms.killQuery('query-502')
```

结果

```
+-------------+-------------+
| queryId     | username    |
+-------------+-------------+
| "query-502" | "joesmith"  |
+-------------+-------------+
1 row
```

5.2.4 因果集群监控相关方法

本节包括可用于监控 Neo4j 因果集群的其他功能。除了前面章节中介绍的监控功能之外，Neo4j 因果集群为操作员提供了监视集群基础功能和状态新特性。这些方法可以一起用于检查集群状态和拓扑情况。

5.2.4.1 查询因果集群核心服务器

可以在因果集群中的每个实例上调用过程 dbms.cluster.role()以返回该实例的角色。语法为：CALL dbms.cluster.role()，返回值为：LEADER、FOLLOWER、READ_REPLICA 三个中的一个，类型为 String。这个函数很有用，是其他监测程序的基础。以下示例显示如何找出当前实例的角色，在本例中为“FOLLOWER”。

```
CALL dbms.cluster.role()
```

结果

```
role
FOLLOWER
```

5.2.4.2 获取集群实例的整体概况

函数 dbms.cluster.overview()能返回集群中所有实例的详细信息，以便获取集群的拓扑概述。语法为：CALL dbms.cluster.overview()，返回值如表 5-27 所示。

表 5-27 获取集群实例整体概况的返回值

名称	类型	描述
Id	String	实例 id
addresses	List<String>	所有实例地址列表
role	String	该实例的角色值为：LEADER、FOLLOWER 或 READ_REPLICA

此函数只能在核心实例上调用，因为核心实例才具有因果集群的完整视图。以下示例显示如何展现集群拓扑，如表 5-28 所示。

表 5-28　CALL dbms.cluster.overview()

Id	addresses	role
08eb9305-53b9-4394-9237-0f0d63bb05d5	[bolt://neo20:7687, http://neo20:7474, https://neo20:7473]	LEADER
cb0c729d-233c-452f-8f06-f2553e08f149	[bolt://neo21:7687, http://neo21:7474, https://neo21:7473]	FOLLOWER
ded9eed2-dd3a-4574-bc08-6a569f91ec5c	[bolt://neo22:7687, http://neo22:7474, https://neo22:7473]	FOLLOWER
00000000-0000-0000-0000-000000000000	[bolt://neo34:7687, http://neo34:7474, https://neo34:7473]	READ_REPLICA
00000000-0000-0000-0000-000000000000	[bolt://neo28:7687, http://neo28:7474, https://neo28:7473]	READ_REPLICA
00000000-0000-0000-0000-000000000000	[bolt://neo31:7687, http://neo31:7474, https://neo31:7473]	READ_REPLICA

5.2.4.3　获取路由推荐

从应用程序的角度来看，它并不关心成员在集群中扮演的角色。相反，应用程序需要知道哪个实例可以提供所需的服务。过程 dbms.cluster.routing.getServers()能提供这些信息，其语法为：CALL dbms.cluster.routing.getServers()，该过程返回特定服务（READ、WRITE 和 ROUTE）与提供此服务的实例的地址之间的映射，还返回生存时间（Time To Live，TTL）信息。TTL 的具体内容类似如下：

```
ttl: 300,
server: [
{
    addresses: [neo20:7687],
    role: WRITE
}, {
    addresses: [neo21:7687, neo22:7687, neo34:7687, neo28:7687, neo31:7687],
    role: READ
}, {
    addresses: [neo20:7687, neo21:7687, neo22:7687],
    role: ROUTE
}
]
```



5.3 安全管理

安全管理的范围比较广泛，对于 Neo4j 数据库而言，首要的是数据安全，可通过遵循有关服务器和网络安全性的行业做法确保物理数据安全，再通过适当的身份验证和授权规则来确保 Neo4j 的信息安全。在“5.3.1 Neo4j 社区版用户管理”小节中仅针对 Neo4j 社区版介绍其用户管理功能，后续“5.3.2 认证和授权”小节、“5.3.3 与 LDAP 集成”小节、“5.3.4 子图访问控制”小节仅适用于 Neo4j 企业版。在“5.3.2 认证和授权”小节中介绍 Neo4j 企业版中的认证和授权，在“5.3.5 安全清单”小节中提供了 Neo4j 安全性的简要清单。

5.3.1 Neo4j 社区版用户管理

Neo4j 社区版在安全管理方面相比企业版而言，功能较弱，仅提供用户和密码管理，没有涉及角色、权限控制等企业必需的安全管理功能。本节将单独针对 Neo4j 社区版用户管理展开详细介绍。在 Neo4j 社区版中，本地用户和角色管理通过使用内置的 Cypher 过程进行管理。本节列出了用户管理的所有安全程序以及一些简单示例，可以使用 Neo4j 浏览器或 Neo4j Cypher Shell 运行下面的示例，除非特别说明，表 5-29 已列出用户管理的全部函数。

表 5-29 用户管理函数列表

序号	名称	描述
1	dbms.security.listUsers	列出所有用户
2	dbms.security.changePassword	改变当前用户密码
3	dbms.security.showCurrentUser	显示当前用户详情
4	dbms.security.createUser	添加一个用户
5	dbms.security.deleteUser	删除一个用户

1. dbms.security.listUsers

此函数的作用是：当前用户查看系统中所有用户的详细信息。语法为：CALL dbms.security.listUsers()，返回值为：username 和 flags，其中 username 为用户名，类型为字符串（String）；flags 表明用户是否需要改变其密码，类型为字符串列表（List<String>）。下面示例显示 Neo4j 社区版中注册的所有用户名以及该用户是否需要更改其密码。

```
CALL dbms.security.listUsers()
```

结果

```
+----------+-------------------------------+
| username | flags                         |
+----------+-------------------------------+
| "neo4j"  | []                            |
| "anne"   | ["password_change_required"] |
| "bill"   | []                            |
+----------+-------------------------------+
3 rows
```



2. dbms.security.changePassword

此函数的作用是：当前用户更改自己的密码。语法为：CALL dbms.security.changePassword(password)，其中的参数 password 为当前用户的新密码，类型为字符串（String），此函数在参数 password 为空或与该用户之前的密码相同时将引发异常。例如要将当前用户的密码更改为“h6u4%kr”，操作如下：

```
CALL dbms.security.changePassword('h6u4%kr')
```

3. dbms.security.showCurrentUser

此函数的作用是：显示当前用户详情，并显示是否需要更改密码。语法为：CALL dbms.security.showCurrentUser()，返回值为：username 和 flags，其中 username 为用户名，类型为字符串（String）；flags 表明用户是否需要改变其密码，类型为：字符串列表（List<String>）。以下示例显示当前用户：用户名为“johnsmith”，不需要更改其密码。

```
CALL dbms.security.showCurrentUser()
```

结果

```
+-------------+-------+
| username    | flags |
+-------------+-------+
| "johnsmith" | []    |
+-------------+-------+
1 row
```

4. dbms.security.createUser

此函数的作用是：将用户添加到 Neo4j 社区版系统。语法为：CALL dbms.security.createUser(username, password, requirePasswordChange)，其中 username 和 password 为新添加用户的用户名和密码，均为字符串类型（String）.，参数 requirePasswordChange 为布尔类型，表明是否需要修改密码，为可选项，默认值为 true，若为 false 值表明：不需要修改密码；若为 true 值表明：该新用户第一次登录时需强制性地改变其密码，如果不修改密码则禁止执行其他任何操作。此函数在下面三种情况下均引发异常：（1）用户名包含字母数字字符以外的字符和“_”字符；（2）用户名已在系统中使用；（3）密码是空字符串。下面示例创建一个用户名为“johnsmith”和密码为“h6u4%kr”的用户，当用户“johnsmith”第一次登录时，需要更改其密码。

```
CALL dbms.security.createUser('johnsmith', 'h6u4%kr', true)
```

5. dbms.security.deleteUser

此函数的作用是：从系统中永久删除用户。语法为：CALL dbms.security.deleteUser(username)，参数 username 为需要被删除的用户名。该函数在下面两种情况下引发异常：（1）用户名在系统中不存在；（2）用户名与当前用户匹配（即不允许删除当前用户）。需要注意：删除用户将立即终止该用户的所有会话，并回滚任何正在运行的事务；由于当前用户不可能删除自己，系统中始终至少有一个用户。以下演示删除用户名



为“janebrown”的用户：

```
CALL dbms.security.deleteUser('janebrown')
```

5.3.2 认证和授权

本小节认证和授权功能仅适用 Neo4j 企业版。Neo4j 中的安全性由身份验证和授权控制两部分组成，认证是确保用户是合法用户的过程，授权涉及检查认证用户是否被允许执行某个动作，使用基于角色的访问控制（Role-Based Access Control，RBAC）管理方式。Neo4j 安全模型的核心是围绕四个预定义的全局数据访问角色：reader、publisher、architect 和 admin。每个角色包括 Neo4j 数据图及其模式（Schema）上允许的一组授权操作，用户可以分配一个或多个角色以及其他自定义角色。有三种方式来进行用户身份验证和授权：本机授权提供方式（Native auth provider）、LDAP 身份验证提供方式（LDAP auth provider）、自定义插件授权提供方式（Custom-built plugin auth provider）。

Neo4j 本机授权提供方式在本地磁盘上存储用户和角色信息，详情参见“5.3.2.1 本机用户和角色管理”小节。LDAP 身份验证提供方式是通过外部安全软件，如 Active Directory 或 OpenLDAP，通过内置的 LDAP 连接器来进行访问，详情参见“5.3.3 与 LDAP 集成”小节。

如果不满足本机授权或 LDAP 身份验证的客户端，Neo4j 还支持自定义插件授权方式，提供身份验证和授权插件的接口，该插件可以访问 neo4j.conf 配置文件，以便加载所需的自定义设置，同时，插件也可写入安全事件日志。具体开发包需要用到 SPI 包，位于 org.neo4j.server.security.enterprise.auth.plugin.spi 路径下，认证插件采用 authenticate 方法来重载实现 AuthPlugin 接口。下面示例为一个最小身份验证插件，使用 neo4j 密码来验证 neo4j 用户。

```
@Override
public AuthenticationInfo authenticate( AuthToken authToken )
{
    String principal = authToken.principal();
    char[] credentials = authToken.credentials();
    if ( principal.equals( "neo4j" ) && Arrays.equals( credentials,
"neo4j".toCharArray() ) )
    {
        return (AuthenticationInfo) () -> "neo4j";
    }
    return null;
}
```

授权插件使用 authorize 方法来重载实现 AuthPlugin 接口。下面示例为一个最小授权插件，它将 reader 角色分配给名为 neo4j 的用户。需注意辅助类 PredefinedRole 的用法。

```
@Override
public AuthorizationInfo authorize( Collection<PrincipalAndProvider>
principals )
{
```

```
    if ( principals.stream().anyMatch( p ->
"neo4j".equals( p.principal() ) ) )
    {
        return (AuthorizationInfo) () ->
Collections.singleton( PredefinedRoles.READER );
    }
    return null;
}
```

还有一个简化的组合插件接口，在 authenticateAndAuthorize 方法中提供身份验证和授权功能。下面示例演示组合验证、授予 reader 角色插件开发：

```
@Override
public AuthInfo authenticateAndAuthorize( AuthToken authToken )
{
    String principal = authToken.principal();
    char[] credentials = authToken.credentials();
    if ( principal.equals( "neo4j" ) && Arrays.equals( credentials,
"neo4j".toCharArray() ) )
    {
        return AuthInfo.of( "neo4j",
Collections.singleton( PredefinedRoles.READER ) );
    }
    return null;
}
```

Neo4j 是一个具有高可扩展性的平台，能轻松满足某些部署可能无法通过标准 LDAP 连接器进行配置的场景。假定 Neo4j 已与 LDAP 用户目录集成，其中组作为用户成员，下面示例先搜索用户所属的组，然后调用自定义的 getNeo4jRoleForGroupId 方法将组映射到 Neo4j 角色：

```
@Override
public AuthInfo authenticateAndAuthorize( AuthToken authToken ) throws
AuthenticationException
{
    try
    {
        String username = authToken.principal();
        char[] password = authToken.credentials();
        LdapContext ctx = authenticate( username, password );
        Set<String> roles = authorize( ctx, username );
        return AuthInfo.of( username, roles );
    }
    catch ( NamingException e )
    {
        throw new AuthenticationException( e.getMessage() );
    }
}
private LdapContext authenticate( String username, char[] password ) throws
NamingException
```

```
{
    Hashtable<String,Object> env = new Hashtable<>();
    env.put( Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.ldap.LdapCtxFactory" );
    env.put( Context.PROVIDER_URL, "ldap://0.0.0.0:10389" );
    env.put( Context.SECURITY_PRINCIPAL,
String.format( "cn=%s,ou=users,dc=example,dc=com", username ) );
    env.put( Context.SECURITY_CREDENTIALS, password );
    return new InitialLdapContext( env, null );
}
private Set<String> authorize( LdapContext ctx, String username ) throws
NamingException
{
    Set<String> roleNames = new LinkedHashSet<>();
    // Set up our search controls
    SearchControls searchCtls = new SearchControls();
    searchCtls.setSearchScope( SearchControls.SUBTREE_SCOPE );
    searchCtls.setReturningAttributes( new String[]{GROUP_ID} );
    // Use a search argument to prevent potential code injection
    Object[] searchArguments = new Object[]{username};
    // Search for groups that has the user as a member
    NamingEnumeration result = ctx.search( GROUP_SEARCH_BASE,
GROUP_SEARCH_FILTER, searchArguments, searchCtls );
    if ( result.hasMoreElements() )
    {
        SearchResult searchResult = (SearchResult) result.next();
        Attributes attributes = searchResult.getAttributes();
        if ( attributes != null )
        {
            NamingEnumeration attributeEnumeration = attributes.getAll();
            while ( attributeEnumeration.hasMore() )
            {
                Attribute attribute = (Attribute) attributeEnumeration.next();
                String attributeId = attribute.getID();
                if ( attributeId.equalsIgnoreCase( GROUP_ID ) )
                {
                    //找到该用户所属的组，检查是否有角色授权。
                    String groupId = (String) attribute.get();
                    String neo4jGroup = getNeo4jRoleForGroupId( groupId );
                    if ( neo4jGroup != null )
                    {
                        //将其添加至角色中。
                        roleNames.add( neo4jGroup );
                    }
                }
            }
        }
    }
    return roleNames;
}
```

自定义插件授权开发的详细信息及示例，可访问官网[19]。

5.3.2.1　本机用户和角色管理

Neo4j 提供本地身份验证，在本地磁盘上存储用户和角色信息，使用这种方式可以完全管理用户。Neo4j 为管理员提供了如下操作：

- 查看所有用户和角色（包括查看用户的所有角色并查看角色的所有用户）。
- 创建和删除用户。
- 从用户分配和删除角色。
- 创建和删除自定义角色。
- 暂停并激活用户。
- 更改用户密码。

当管理员暂停或删除其他用户时，可进行如下操作：

- 管理员可以暂停或删除任何其他用户（包括其他管理员），但不能是自己本身。
- 删除用户将终止该用户正在运行的查询和会话。
- 当前为已删除的用户，其运行的所有查询都将回滚。
- 用户将无法再次登录（如果暂停的话，则直到管理员重新激活）。
- 在删除用户之前，无须从用户中删除分配的角色。

Neo4j 在基于角色的访问控制框架中提供了四个内置角色：

- reader：只读角色，只读访问图数据库（包括所有节点、关系和属性）。
- publisher：发布者角色，对图数据有读写访问权限。
- architect：架构师角色，除对图数据可读写访问外，还能够设置/删除索引以及任何其他将来设计的访问模式。
- admin：超级管理员角色，包括 architect 角色所有权限，并能查看/终止查询。

表 5-30 详细规定了每个角色对数据和数据库所能进行的操作集合。

表 5-30　角色操作列表

操作	reader	publisher	architect	admin	（无角色）
更改自己的密码	√	√	√	√	√
查看自己的详细信息	√	√	√	√	√
读取数据	√	√	√	√	
查看自己的查询	√	√	√	√	
终止自己的查询	√	√	√	√	
写/更新/删除数据		√	√	√	
创建/删除索引/约束			√	√	
创建/删除用户				√	
更改其他用户的密码				√	
向/从用户分配/删除角色				√	

[19] https://github.com/neo-technology/neo4j-example-auth-plugins

（续表）

操作	reader	publisher	architect	admin	（无角色）
挂起/激活用户				√	
查看所有用户/角色				√	
查看用户的所有角色				√	
查看角色的所有用户				√	
查看所有查询				√	
终止所有查询				√	

没有分配角色的用户将不具有数据的任何权限，甚至包括：不具有读取权限。一个用户可以分配多个角色，其权限为这多个角色操作的并集。

Neo4j 除了上述系统内置的四种角色（Reader、Publisher、Architect、Admin）外，还支持自定义角色（Custom Roles），它是由数据库管理员（Administrator，即赋予 admin 权限的用户）创建或删除。创建自定义角色的唯一目的是控制执行某些自定义的开发过程，与内置角色相比，自定义角色不具备任何权限，只能执行在 neo4j.conf 中明确规定的过程。有关如何使用自定义角色来控制子图访问的更多细节，请参见“5.3.4 子图访问控制”小节。

5.3.2.2 用户和角色的传播

本节介绍如何在 Neo4j 集群中传播本机用户、角色和角色的分配。本机用户、角色和角色分配存储在名为 auth 和 roles 的文件中，位于 dbms 的子目录的 Data 目录中，Neo4j 每 5 秒自动从磁盘重新加载用户和角色信息，对用户和角色的更改仅适用于执行命令的 Neo4j 实例，这意味着更改不会自动传播到 Neo4j 实例的集群中，除非单独指定。将本机用户、自定义角色和角色分配的更改信息传播给集群相关的方法有如下三种：

（1）将磁盘上的用户和角色文件手动复制到所有其他集群实例。
（2）使用共享网络文件夹存储用户和角色文件。
（3）创建一个自动过程，比如：结合使用 rsync、crontab 来同步集群中存储的数据。

为了集群的整体安全性，建议使用 LDAP 或插件授权提供方式。

5.3.2.3 本机用户和角色管理的过程

本小节介绍 Neo4j 企业版中本机用户和角色管理相关的过程。在 Neo4j 企业版中，本机用户和角色管理通过使用内置的 Cypher 过程进行管理。与 Neo4j 社区版相比，Neo4j 企业版的安全管理主要是增加了角色管理，包括内置的四种角色（reader、publisher、architect、admin）和自定义角色（Custom roles），能更好地支持用户分类管理和分类授权，为企业级应用奠定安全管理的基础。

显然，在内置安全管理过程方面，与 Neo4j 社区版用户管理表 5-29 列出来的 5 个用户管理过程（listUsers、changePassword、showCurrentUser、CreateUser、deleteUser）相比，明显增多，Neo4j 企业版中增至 15 个。其中有 4 个过程（listUsers、changePassword、CreateUser、deleteUser）在社区版和企业版中均有，但在企业版中调用这些过程时，要求其具备管理员角色权限，而在社区版中没有角色这一功能。显然企业版与社区版在用户管理过程方面，还是有一定量的差异，比如：listUsers 在返回值方面就新增加一列“roles”，即角

色等。建议读者依据所使用的版本不同来正确使用这些安全管理功能。

下面将列出 Neo4j 企业版用户管理的所有安全程序以及一些简单示例的列表，可以使用 Neo4j 浏览器或 Neo4j Cypher Shell 来运行所提供的示例，主要包括 15 个内置 Cypher 过程，简要信息如表 5-31 所示。

表 5-31　内置 Cypher 过程列表

序号	过程名	要求权限	中文说明
1	dbms.security.listUsers()	管理员	列出所有用户
2	dbms.security.listRoles()	管理员	列出所有角色及该角色下的所有用户
3	dbms.security.listRolesForUser(username)		列出指定用户名下的角色，如果为管理员，则列出系统中所有角色
4	dbms.security.listUsersForRole(roleName)	管理员	列出赋予该角色的所有用户
5	dbms.security.createUser(username, password, requirePasswordChange)	管理员	创建新用户
6	dbms.security.deleteUser(username)	管理员	删除用户
7	dbms.security.addRoleToUser(roleName, username)	管理员	给指定用户增加角色
8	dbms.security.removeRoleFromUser(roleName, username)	管理员	给指定用户删除角色
9	dbms.security.createRole(roleName)	管理员	创建自定义角色
10	dbms.security.deleteRole(roleName)	管理员	删除自定义角色
11	dbms.security.suspendUser(username)	管理员	挂起指定用户
12	dbms.security.activateUser(username, requirePasswordChange)	管理员	激活指定用户
13	dbms.security.changeUserPassword(username, newPassword, requirePasswordChange)		改变用户密码，管理员可以改变任何用户密码，当前用户只能改变自己的密码
14	dbms.security.changePassword(password, requirePasswordChange)		改变当前用户密码
15	dbms.procedures()	管理员	列出每个过程所需的角色

下面分别就每个过程逐一进行详细阐述，并附上示例说明。

1. dbms.security.listUsers

此过程只能是管理员查看系统中每个用户的详细信息。语法为：CALL dbms.security.listUsers()。返回三个值，如表 5-32 所示。

表 5-32　listUsers 过程返回值

名称	类型	说明
username	String	用户名
roles	List<String>	该用户赋予的角色名列表
flags	List<String>	标志列表，表明用户是否挂起或需要更改其密码

该过程调用的当前用户如果不是管理员，则引发异常。下面示例，显示系统中的用户名、分配给用户的角色，及用户是否挂起或需要更改其密码。

```
CALL dbms.security.listUsers()
```

结果

```
+----------+---------------------------+-----------------------------+
| username | roles                     | flags                       |
+----------+---------------------------+-----------------------------+
| "neo4j"  | ["admin"]                 | []                          |
| "anne"   | []                        | ["password_change_required"] |
| "bill"   | ["reader"]                | ["is_suspended"]            |
| "john"   | ["architect","publisher"] | []                          |
+----------+---------------------------+-----------------------------+
4 rows
```

2. dbms.security.listRoles

此过程由管理员调用显示每个角色被分配的用户列表。语法为：CALL dbms.security.listRoles()。返回两个值，如表 5-33 所示。

表 5-33 listRoles 过程返回值

名称	类型	说明
Role	String	角色名
Users	List<String>	用户列表，表明该角色被分配的用户列表

该过程调用的当前用户如果不是管理员，则引发异常。下面的示例显示每个角色和已分配用户列表。

```
CALL dbms.security.listRoles()
```

结果

```
+-------------+----------------+
| role        | users          |
+-------------+----------------+
| "reader"    | ["bill"]       |
| "architect" | []             |
| "admin"     | ["neo4j"]      |
| "publisher" | ["john","bob"] |
+-------------+----------------+
4 rows
```

3. dbms.security.listRolesForUser

此过程显示用户所赋予的角色，任何活跃用户（除挂起用户外）都可以调用，管理员可以查看任何用户所赋予的角色。语法为：CALL dbms.security.listRolesForUser(username)，其中参数 username 为需要查看的用户名，返回值为 String 类型，为该用户所赋予的角色。调用此过程可能引起异常的情况有两种：当前使用用户不是管理员，但查看的用户是管理员；查看的用户名在系统中不存在。需要注意：该过程可以由当前用户调用以查看自身的角色，管理员可以调用此过程以查看另一个用户的角色。以下示例列出用户名为“johnsmith”的所有角色，其角色为 reader 和 publisher。

```
CALL dbms.security.listRolesForUser('johnsmith')
```

结果

```
+-------------+
| value       |
+-------------+
| "reader"    |
| "publisher" |
+-------------+
2 rows
```

4. dbms.security.listUsersForRole

此过程由管理员调用显示某个角色被赋予的所有用户。语法为：CALL dbms.security.listUsersForRole(roleName)，参数 roleName 为角色名，返回值为 String 类型，为该角色被赋予的所有用户。调用此过程可能引起异常的情况有两种：当前用户不是管理员；或角色名在系统中不存在。以下示例列出了赋予 publisher 角色的所有用户 bill、anne。

```
CALL dbms.security.listUsersForRole('publisher')
```

结果

```
+--------+
| value  |
+--------+
| "bill" |
| "anne" |
+--------+
2 rows
```

5. dbms.security.createUser

此过程由管理员调用来创建新用户。一般会接着给用户分配一个角色。语法为：CALL dbms.security.createUser(username, password, requirePasswordChange)，三个参数的描述如表 5-34 所示。

表 5-34　createUser 过程返回值

名称	类型	说明
username	String	新建的用户名
password	String	新建用户的密码
requirePasswordChange	Boolean	可选项，默认值为 true。表明：是否要求用户更改密码。如果为 true，则用户首次登录系统要求其修改密码；如果不修改，则禁止执行任何其他操作

调用此过程可能引起异常的情况有 4 种：当前用户不是管理员；用户名包含字母数字字符以外的字符或“_”字符；用户名已在系统中使用；密码是空字符串。以下示例创建一个用户名为“johnsmith”和密码为“h6u4%kr”的用户，当用户“johnsmith”第一次登录时，系统要求其更改密码。

```
CALL dbms.security.createUser('johnsmith', 'h6u4%kr')
```

6. dbms.security.deleteUser

此过程由管理员从系统中永久删除某个用户，此操作一旦执行就无法撤消，因此，如果

仅供非商业用途或交流学习使用

只是对用户有疑问，可以考虑暂停该用户。语法为：CALL dbms.security.deleteUser(username)，参数 username 为 String 类型，是需要删除的用户名。调用此过程可能引起异常的情况有三种：当前用户不是管理员；用户名在系统中不存在；用户名与当前用户匹配（即不允许删除当前用户）。需要注意：在删除用户之前，不必从用户中删除任何分配的角色；删除用户将立即终止其所有用户的会话，并回滚任何正在运行的事务；由于当前用户不能删除自己，系统中始终至少有一个管理员。以下示例删除用户名为“janebrown”的用户。

```
CALL dbms.security.deleteUser('janebrown')
```

7. dbms.security.addRoleToUser

此过程由管理员给用户添加角色，使其具备相应的数据操作权限。语法为：CALL dbms.security.addRoleToUser(roleName, username)，其中参数 roleName 为角色名，参数 username 为用户名。调用此过程可能引起异常的情况有 4 种：当前用户不是管理员；用户名在系统中不存在；用户名包含字母数字字符以外的字符和“_”字符；角色名称不存在于系统中；角色名称包含字母数字字符以外的字符和“_”字符。注意：此过程操作具有幂等性，即多次调用与调用一次效果相同。以下示例将 publisher 角色分配给 johnsmith 用户。

```
CALL dbms.security.addRoleToUser('publisher', 'johnsmith')
```

8. dbms.security.removeRoleFromUser

此过程由管理员对用户移除角色，从而防止用户对数据执行由此角色规定的操作。语法为：CALL dbms.security.removeRoleFromUser(roleName, username)，其中参数 roleName 为角色名，参数 username 为用户名。调用此过程可能引起异常的情况有四种：当前用户不是管理员；用户名不存在；角色名不存在；删除的用户为当前用户且角色是 admin。需要注意：如果 username 是当前用户，并且 roleName 是 admin，则会报错，即当前用户不可以降级为管理员，当前用户不能自己删除管理员角色，即系统中始终至少有一个管理员。该过程同样具有幂等性。以下示例将 johnsmith 用户删除 publisher 角色。

```
CALL dbms.security.removeRoleFromUser('publisher', 'johnsmith')
```

9. dbms.security.createRole

此过程由管理员创建自定义角色。语法为：CALL dbms.security.createRole(roleName)，参数 roleName 为 String 类型，是需要新创建的角色名称。调用此过程可能引起异常的情况有 5 种：当前用户不是管理员；角色名称已存在；角色名称为空；角色名称包含字母数字字符以外的字符或“_”字符；角色名称匹配其中一个本地内置角色：reader、publisher、architect 和 admin。以下示例创建一个新的自定义 operator 角色。

```
CALL dbms.security.createRole('operator')
```

10. dbms.security.deleteRole

此过程由管理员从系统中删除某个自定义角色，但无法删除系统内置的 4 个角色：

仅供非商业用途或交流学习使用

reader、publisher、architect 和 admin。语法为：CALL dbms.security.deleteRole(roleName)，参数 roleName 为 String 类型，是需要删除的角色名称。调用此过程可能引起异常的情况有三种：当前用户不是管理员；角色名称不存在；角色名称为内置角色：reader、publisher、architect 和 admin。需要注意：调用此过程，删除自定义角色，它所被分配的任何用户将不要拥有该角色所定义的任何操作。以下示例从系统中删除自定义 operator 角色。

```
CALL dbms.security.deleteRole('operator')
```

11. dbms.security.suspendUser

此过程由管理员调用将某用户从系统中挂起，暂停的用户可以在后续被激活。语法为：CALL dbms.security.suspendUser(username)，其中参数 username 为 String 类型，是需要挂起的用户名称。调用此过程可能引起异常的情况有三种：当前用户不是管理员；用户名不存在；用户名与当前用户的名称匹配（即不允许挂起当前用户）。需要注意：挂起用户将立即终止该用户的所有会话并回滚其正在运行的事务；挂起用户的属性（分配的角色和密码）保持不变；挂起的用户将无法登录到系统；由于当前用户不可能挂起自己，系统中将始终至少有一个活动的管理员用户。此过程同样具有幂等性。以下示例挂起 billjones 用户。

```
CALL dbms.security.suspendUser('billjones')
```

12. dbms.security.activateUser

此过程由管理员激活某个挂起的用户，以便用户再次能够访问数据库。语法为：CALL dbms.security.activateUser(username, requirePasswordChange)，参数 username 为激活的用户名，参数 requirePasswordChange 为布尔类型，默认值为 true，表明是否要求此用户登录时修改密码。调用此过程可能引起异常的情况有三种：当前用户不是管理员；用户名不存在；用户名与当前用户的名称匹配（即不允许激活当前用户）。此过程同样具有幂等性。以下示例激活 jackgreen 用户，且下次登录时，需要更改其密码。

```
CALL dbms.security.activateUser('jackgreen')
```

13. dbms.security.changeUserPassword

此过程为更改用户密码，管理员可以更改系统中任何用户的密码，当前用户仅能改变自己的密码。语法为：CALL dbms.security.changeUserPassword(username, newPassword, requirePasswordChange)，其中参数 username 为用户名，参数 newPassword 为新密码，参数 requirePasswordChange 为布尔类型，默认值为 true，表明是否要求此用户登录时修改密码。调用此过程可能引起异常的情况有 4 种：当前用户不是管理员，并且用户名与当前用户的用户名不匹配；用户名不存在；密码是空字符串；密码与用户以前的密码相同。

需要注意：该过程可以由当前用户调用以改变自己的密码，而不管当前用户是否为管理员；如果管理员调用则可以更改其他用户的密码；除了更改用户的密码，还将立即终止该用户的会话，并回滚该用户正在运行的事务。以下示例将 joebloggs 用户的密码更改为“h6u4%kr”，且下次登录时，系统要求更改其密码。

仅供非商业用途或交流学习使用

```
CALL dbms.security.changeUserPassword('joebloggs', 'h6u4%kr')
```

14. dbms.security.changePassword

此过程调用为当前用户更改密码。语法为：CALL dbms.security.changePassword(password, requirePasswordChange)，其中 password 参数为新密码，参数 requirePasswordChange 为布尔类型，默认值为 true，表明是否要求此用户登录时修改密码。调用此过程可能引起异常的情况有两种：密码是空字符串；密码与当前用户的旧密码相同。以下示例将当前用户的密码更改为“h6u4%kr”。

```
CALL dbms.security.changePassword('h6u4%kr')
```

15. dbms.procedures

此过程由管理员列出系统中的所有过程，包括角色权限等。语法为：CALL dbms.procedures()，其返回值简要说明如表 5-35 所示。

表 5-35 procedures 过程返回值

名称	类型	说明
name	String	过程名
signature	String	过程标示
description	String	过程描述
roles	List<String>	具有执行此过程的角色列表

如果是非管理员调用此过程将引发异常。以下示例显示了包含“security”关键字的 3 个过程，包括：过程名称、描述以及哪些角色可以执行此过程。

```
CALL dbms.procedures()
YIELD name, signature, description, roles
WITH name, description, roles
WHERE name contains 'security'
RETURN name, description, roles
ORDER BY name
LIMIT 4
```

结果

```
+---------------------------------+-----------------------------------+---------+
| name                            | description                       | roles   |
+---------------------------------+-----------------------------------+---------+
|"dbms.security.activateUser"     |"Activate a suspended user."       |["admin"]|
|"dbms.security.addRoleToUser"    |"Assign a role to the user."       |["admin"]|
|"dbms.security.changePassword"   |"Change the current user's password."|["admin"]|
|"dbms.security.changeUserPassword"|"Change the given user's password."|["admin"]|
+---------------------------------+-----------------------------------+---------+
4 rows
```

5.3.3 与 LDAP 集成

LDAP 以 X.500 标准为基础，但与 X.500 相比 LDAP 更简单，它可根据需要定制，并支持 TCP/IP 协议，因此使用非常广泛。本小节将介绍 Neo4j 与 LDAP 协议的集成。

仅供非商业用途或交流学习使用

Neo4j 本身支持 LDAP 协议，包括：Active Directory、OpenLDAP 或其他 LDAP 兼容的身份验证服务。下面示例对 Neo4j 数据库进行相应的配置，其中用户管理采用 LDAP 服务，这意味着完全关闭本机 Neo4j 用户和角色管理，并将 LDAP 组映射到 4 个内置 Neo4j 角色 Creader、publisher、architect 和 admin）和自定义角色。所有设置都需要在服务器启动时在默认配置文件 neo4j.conf 中进行设定，neo4j.conf 配置详见“附录 A Neo4j 配置设置”。首先配置 Neo4j 使用 LDAP 作为身份验证和授权提供方。

```
# Turn on security
dbms.security.auth_enabled=true
# Choose LDAP connector as security provider for both authentication and
authorization
dbms.security.auth_provider=ldap
```

接下来配置活动目录（Active Directory），示例如下：

```
# Configure LDAP to point to the AD server
dbms.security.ldap.host=ldap://myactivedirectory.example.com
# Provide details on user structure within the LDAP system:
dbms.security.ldap.authentication.user_dn_template=cn={0},cn=Users,dc=exampl
e,dc=com
dbms.security.ldap.authorization.user_search_base=cn=Users,dc=example,dc=com
dbms.security.ldap.authorization.user_search_filter=(&(objectClass=*)(cn={0}
))
dbms.security.ldap.authorization.group_membership_attributes=memberOf
# Configure the actual mapping between groups in the LDAP system and roles
in Neo4j
dbms.security.ldap.authorization.group_to_role_mapping=\
  "cn=Neo4j Read Only,cn=Users,dc=neo4j,dc=com"      = reader      ;\
  "cn=Neo4j Read-Write,cn=Users,dc=neo4j,dc=com"     = publisher   ;\
  "cn=Neo4j Schema Manager,cn=Users,dc=neo4j,dc=com" = architect   ;\
  "cn=Neo4j Administrator,cn=Users,dc=neo4j,dc=com"  = admin       ;\
  "cn=Neo4j Procedures,cn=Users,dc=neo4j,dc=com"     = allowed_role
# In case defined users are not allowed to search for themselves,
# we can specify credentials for a user with read access to all users and
groups:
# Note that this account only needs read-only access to the relevant parts
of the LDAP directory
# and does not need to have access rights to Neo4j or any other systems.
#dbms.security.ldap.authorization.use_system_account=true
#dbms.security.ldap.authorization.system_username=cn=search-
account,cn=Users,dc=example,dc=com
#dbms.security.ldap.authorization.system_password=secret
```

再以 openLDAP 为示例，配置如下：

```
# Configure LDAP to point to the OpenLDAP server
dbms.security.ldap.host=myopenldap.example.com
# Provide details on user structure within the LDAP system:
dbms.security.ldap.authentication.user_dn_template=cn={0},ou=users,dc=exampl
e,dc=com
```


仅供非商业用途或交流学习使用

```
dbms.security.ldap.authorization.user_search_base=ou=users,dc=example,dc=com
dbms.security.ldap.authorization.user_search_filter=(&(objectClass=*)(uid={0
}))
dbms.security.ldap.authorization.group_membership_attributes=gidnumber
# Configure the actual mapping between groups in the OpenLDAP system and
roles in Neo4j
dbms.security.ldap.authorization.group_to_role_mapping=\
  101 = reader      ;\
  102 = publisher   ;\
  103 = architect   ;\
  104 = admin       ;\
  105 = allowed_role
# In case defined users are not allowed to search for themselves,
# we can specify credentials for a user with read access to all users and
groups:
# Note that this account only needs read-only access to the relevant parts
of the LDAP directory
# and does not need to have access rights to Neo4j or any other systems.
#dbms.security.ldap.authorization.use_system_account=true
#dbms.security.ldap.authorization.system_username=cn=search-
account,ou=users,dc=example,dc=com
#dbms.security.ldap.authorization.system_password=search-account-password
```

需要注意配置示例中的一些细节，有关 LDAP 配置的参数如表 5-36 所示。

表 5-36　LDAP 配置参数

参数名	默认值	描述
dbms.security.ldap.authentication.user_dn_template	uid={0},ou=users,dc=example,dc=com	将用户名转换为登录所需的 LDAP 特定全名
dbms.security.ldap.authorization.user_search_base	ou=users,dc=example,dc=com	设置基础对象或命名上下文以搜索用户对象
dbms.security.ldap.authorization.user_search_filter	(&(objectClass=*)(uid={0}))	设置 LDAP 搜索过滤器以搜索用户主体
dbms.security.ldap.authorization.group_membership_attributes	[memberOf]	列出包含要用于映射到角色的组的用户对象上的属性名称
dbms.security.ldap.authorization.group_to_role_mapping		列出从组到预定义的内置角色 admin、architect、publisher、reader 或其他自定义角色的授权映射

可以使用 LDAP 命令行工具 ldapsearch 来验证配置是否正确、LDAP 服务器是否正在响应，可以通过发送包含 LDAP 配置设置值的搜索命令来执行此项操作。示例搜索验证用户“john”的认证（使用简单机制）和授权。

默认情况下 dbms.security.ldap.authorization.use_system_account=false 的验证是：

```
#ldapsearch -v -H ldap://<dbms.security.ldap.host> -x -D
<dbms.security.ldap.authentication.user_dn_template : replace {0}> -W -b
<dbms.security.ldap.authorization.user_search_base>
"<dbms.security.ldap.authorization.user_search_filter : replace {0}>"
<dbms.security.ldap.authorization.group_membership_attributes>
ldapsearch -v -H ldap://myactivedirectory.example.com:389 -x -D
```


仅供非商业用途或交流学习使用

```
cn=john,cn=Users,dc=example,dc=com -W -b cn=Users,dc=example,dc=com
"(&(objectClass=*)(cn=john))" memberOf
```

参数 dbms.security.ldap.authorization.use_system_account=true 的验证为：

```
#ldapsearch -v -H ldap://<dbms.security.ldap.host> -x -D
<dbms.security.ldap.authorization.system_username> -w
<dbms.security.ldap.authorization.system_password> -b
<dbms.security.ldap.authorization.user_search_base>
"<dbms.security.ldap.authorization.user_search_filter>"
<dbms.security.ldap.authorization.group_membership_attributes>
ldapsearch -v -H ldap://myactivedirectory.example.com:389 -x -D cn=search-
account,cn=Users,dc=example,dc=com -w secret -b cn=Users,dc=example,dc=com
"(&(objectClass=*)(cn=john))" memberOf
```

接下来验证响应是否成功，且 dbms.security.ldap.authorization.group_to_role_mapping 中的角色组与成员资格属性的值映射正确。

```
# extended LDIF
# LDAPv3
# base <cn=Users,dc=example,dc=com> with scope subtree
# filter: (cn=john)
# requesting: memberOf
# john, Users, example.com
dn: CN=john,CN=Users,DC=example,DC=com
memberOf: CN=Neo4j Read Only,CN=Users,DC=example,DC=com
# search result
search: 2
result: 0 Success
# numResponses: 2
```

认证缓存（Auth Cache）是 Neo4j 通过 LDAP 服务器缓存认证结果以提升性能的机制，它采用 dbms.security.ldap.authentication.cache_enabled 和 dbms.security.auth_cache_ttl 参数进行配置。示例配置如下：

```
# Turn on authentication caching to ensure performance
dbms.security.ldap.authentication.cache_enabled=true
dbms.security.auth_cache_ttl=10m
```

上述两个参数的具体描述如表 5-37 所示。

表 5-37　认证缓存参数

参数名	默认值	描述
dbms.security.ldap.authentication.cache_enabled	true	确定是否通过 LDAP 服务器缓存身份验证的结果。依据自身安全考虑，决定是否启用验证缓存。需要注意，当使用 REST API 时，禁用认证缓存将导致对每个请求的用户的重新认证和授权，可能严重影响系统的性能，增加 LDAP 服务器的负载
dbms.security.auth_cache_ttl	10000 minutes	缓存身份验证和授权信息的生存时间（Time To Live，TTL）。将 TTL 设置为 0 为禁用验证缓存；TTL 太短将导致频繁重新认证和授权，可能影响性能；TTL 太长则对 LDAP 服务器上用户设置的更改可能不会及时反映在 Neo4j 授权行为中

仅供非商业用途或交流学习使用

管理员可以手动清除身份验证缓存，以强制重新查询来自联合身份验证提供的系统身份验证和授权信息，可以在 Neo4j 浏览器或 Neo4j Cypher Shell 中执行语句：CALL dbms.security.clearAuthCache()即可。

可以通过下列方法来指定 dbms.security.ldap.host 参数，配置 LDAP 协议为非加密方式，如果不指定协议或端口，则采用默认协议为 LDAP，默认端口为 389。

```
dbms.security.ldap.host=myactivedirectory.example.com
dbms.security.ldap.host=myactivedirectory.example.com:389
dbms.security.ldap.host=ldap://myactivedirectory.example.com
dbms.security.ldap.host=ldap://myactivedirectory.example.com:389
```

可以采用 StartTLS 来配置对活动目录的加密，相关设置如下：

```
dbms.security.ldap.use_starttls=true
dbms.security.ldap.host=ldap://myactivedirectory.example.com
```

也可以使用加密 LDAPS 来配置活动目录，只需将 dbms.security.ldap.host 设置为以下某个值，如果不指定端口将使用默认端口 636，默认协议为 LDAPS。但这种方法已被弃用，不推荐使用。推荐采用 StartTLS 来对活动目录加密。

```
dbms.security.ldap.host=ldaps://myactivedirectory.example.com
dbms.security.ldap.host=ldaps://myactivedirectory.example.com:636
```

在实际生产环境中，建议使用由证书颁发机构颁发的 SSL 证书来确保对 LDAP 服务器的安全访问。但是有些情况下，例如在测试环境中，可能希望在 LDAP 服务器上使用自签名证书。可以在 neo4j.conf 中使用参数 dbms.jvm.additional 来指定证书的详细信息，以告知 Neo4j 本地证书的位置。示例如下：

```
dbms.jvm.additional=-Djavax.net.ssl.keyStore=MyCert.jks
dbms.jvm.additional=-Djavax.net.ssl.keyStorePassword=secret
dbms.jvm.additional=-Djavax.net.ssl.trustStore=MyCert.jks
dbms.jvm.additional=-Djavax.net.ssl.trustStorePassword=secret
```

5.3.4 子图访问控制

通过使用用户定义的过程和自定义角色（Custom roles），管理员可以将用户的访问和动作限制到图中指定部分，即：可以在子图的级别配置访问控制。自定义角色是由 Neo4j 数据库管理员创建和删除，仅用于控制执行某些自定义开发的过程，与数据库自带的角色相比，自定义角色的权限需要在 neo4j.conf 中显式准许。例如，可以允许用户读取但不写入标记有 Employee 的节点和类型为 REPORTS_TO 的关系。

5.3.4.1 管理自定义角色

1. 本机用户场景

创建自定义角色，然后将此角色分配给相关用户。例如：创建 accounting 角色并将其分配给预先存在的 billsmith 用户。

仅供非商业用途或交流学习使用

```
CALL dbms.security.createRole('accounting')
CALL dbms.security.addRoleToUser('accounting', 'billsmith')
```

2. 联合用户场景（LDAP）

在 LDAP 方案中，LDAP 用户组必须映射到 Neo4j 中的自定义角色。例如：将组号 101 的 LDAP 组映射到自定义 accounting 角色：

```
dbms.security.realms.ldap.authorization.group_to_role_mapping=101=accounting
```

5.3.4.2　配置过程的权限

需要自行创建读或写数据的过程（Procedure），自然内置或第三方库自带的过程除外。此过程与普通 Cypher 语句执行的安全规则相同，例如，具有写数据权限的过程可以被分配给 publisher、architect、admin 角色，而仅分配 reader 角色的用户将不允许执行此过程。可以使用配置选项 dbms.security.procedures.default_allowed 和 dbms.security.procedures.roles 来准许特定角色执行相应的访问过程，可以设置 dbms.security.procedures.default_allowed 允许角色执行任何与 dbms.security.procedures.roles 配置不匹配的过程。对过程更细粒度的控制可通过 dbms.security.procedures.roles 来进行设置。例如，dbms.security.procedures.roles = apoc.convert.*:Converter;apoc.load.json.*:Converter,DataSource; apoc.trigger.add:TriggerHappy 可以达到如下效果：

- 具有 Converter 角色的用户都可执行 apoc.convert 命名空间中的所有过程。
- 角色 Converter 和 DataSource 可执行 apoc.load.json 命名空间中的过程。
- TriggerHappy 角色可执行 apoc.trigger.add 过程。

过程的角色配置用于重置用户角色所配置的权限，通过特定过程执行的方式来重置用户许可。如果某个过程尝试执行未包含在其模式中的数据库操作，该过程操作将失败并提示“权限被拒绝”的错误，并给出为何禁止执行此过程的原因。过程权限配置相关的参数 dbms.security.procedures.default_allowed、dbms.security.procedures.roles 详细解释，请参见“附录 A Neo4j 配置设置”。

5.3.5　安全清单

安全清单为 Neo4j 数据库安全性建议的总结。特别指出可能需要额外关注，以满足与应用程序相匹配的安全级别。

（1）在安全网络及安全服务器上部署 Neo4j：使用子网和防火墙，只打开必要的端口。

（2）保护静止数据：使用卷加密（例如 Bitlocker）；管理对数据库转储和备份的访问。请确保没有对设置 dbms.backup.address（此默认值为 6362）指定端口的外部访问。如果未保护此端口，将带来安全隐患：未授权的用户可以通过该漏洞将数据库的副本复制到其他计算机上。管理好数据文件和事务日志的访问，按权限设置好操作系统对 Neo4j 数据文件和事务日志文件的访问。

（3）保护传输中数据：对远程访问 Neo4j 数据库，只打开加密的 Bolt 或 HTTPS 访问链

仅供非商业用途或交流学习使用

接，使用受信任证书颁发机构颁发的 SSL 证书。有关在 Neo4j 中安装 SSL 证书的信息，请参见“5.1.8 安装证书”小节；有关配置 Bolt 或 HTTPS 连接的信息，请参见“5.1.7 配置 Neo4j 连接器”小节；如果使用 LDAP，请通过 StartTLS 对 LDAP 系统进行加密，请参见“5.3.3 与 LDAP 集成”小节。

（4）验证部署的任何自定义代码（过程和非托管扩展），并确保它们不会无意中暴露产品或数据的任何部分。

（5）确保 Neo4j 文件正确的文件权限。只有操作系统用户才配置这些文件的权限，有关权限级别的说明，请参见“5.1.2 文件位置”小节。尤其需要保护数据文件、事务日志和数据库转储，防止未经授权的读取访问。通过限制对 bin、lib 和 plugins 目录的访问来防止未经授权扩展的执行。

（6）如果启用了 LOAD CSV 函数操作，请确保未授权用户不可导入数据。

（7）不要关闭 Neo4j 验证。

（8）检查 neo4j.conf 文件，了解已弃用函数（例如由参数 dbms.shell.port 控制的 neo4j-shell）和远程 JMX（由 dbms.jvm.additional=-Dcom.sun.management.jmxremote.port=3637 控制）相关的端口。

（9）使用 Neo4j 最新的补丁版本。

5.4 Neo4j 数据库运维与优化

本节介绍影响操作性能的因素，以及如何调整 Neo4j 以获得最佳吞吐量。

5.4.1 内存调优

本节介绍如何为 Neo4j 实例配置内存，阐明在各种内存需求场景下如何权衡以及垃圾收集的特性。Neo4j 在启动时将自动配置内存相关参数的默认值，并默认可使用机器上的所有内存。有三种类型的内存要考虑：操作系统内存、页面缓存和堆空间。

需要注意的是，操作系统内存是不能显式配置的，而是在指定页面缓存和堆空间后所“剩下的”。如果将页面缓存和堆空间配置为等于或大于可用内存，或者没有为操作系统留出足够的空间，操作系统内存将进行磁盘交换，这将严重影响性能。请确保操作系统内存配置为永远不需要交换。因此，内存调优可按照如下步骤进行：

- 计划操作系统内存大小。
- 计划页面缓存大小调整。
- 计划堆空间大小。
- 做合理的检查：实际操作系统内存分配=可用内存 -（页面缓存+堆空间大小）。

5.4.1.1 操作系统内存大小

必须为服务器上与 Neo4j 数据库不相关的其他程序预留一些内存。此外，还需为 index

仅供非商业用途或交流学习使用

和 schema 目录留出足够的内存用作操作系统的文件缓冲区，因为如果索引不能全部装载在内存中，将影响索引查找的性能。1GB 内存是 Neo4j 服务器的最低配置。基本的计算方法如下：

系统内存=1GB +（graph.db/index）+（graph.db/schema）

5.4.1.2　页面缓存大小

页面缓存用于缓存存储在磁盘上的 Neo4j 数据，确保将来自磁盘的所有或至少大部分图数据缓存到内存中，将有助于避免昂贵的磁盘访问并获得最佳性能。可通过简单的方法确定页面缓存的大小：汇总含 NEO4J_HOME/data/databases/graph.db/*store.db*的所有文件大小，再增加 20%的预留。指定页缓存的参数是：dbms.memory.pagecache.size，该参数设定允许 Neo4j 使用多少内存用于高速缓存，参数详见表 5-38。如果没有在启动时明确指定，Neo4j 将给予默认配置：（机器可用内存 - JVM 最大堆分配）*50%。以下是估计页缓存大小的两种可行的方法：

- 对于现有的 Neo4j 数据库，查询所有*store.db*文件的大小，来确定页面缓存需容纳所有数据的大小，再增加 20%。例如：在 posix 系统中从 data/databases/graph.db 目录下运行$ du -hc *store.db*命令即可获得数据大小。
- 对于新的 Neo4j 数据库，先导入一定比率的数据（例如总数据量的百分之一），然后将得到的存储大小乘以该比率（此处举例为 100），再额外增加 20%。

表 5-38　页面缓存配置参数

参数	可能值	效果
dbms.memory.pagecache.size	用于页面缓存的最大内存量，以字节为单位，或者更大的字节单位，例如 100MB、4GB	用于映射存储文件的内存量，以字节为单位。这将自动向下舍入到最近的整页，此值不能为零。对于极小和内存受限的部署，建议仍为页面缓存至少保留几兆字节
unsupported.dbms.report_configuration	true 或 false	true 表示：写入默认系统输出，主要是控制台或日志文件

5.4.1.3　堆大小

可用堆内存大小是影响 Neo4j 性能的一个重要因素。一般来说，配置足够大的堆空间以维持并发操作是非常有用的。大多数 Neo4j 应用，堆大小设置为 8~16GB 之间即稳定运行。在文件 NEO4J_HOME/conf/neo4j-wrapper.conf 中设置 dbms.memory.heap.initial_size 和 dbms.memory.heap.max_size 参数即可，以兆字节（MB）为单位，比如：16000。建议将这两个参数设置为相同的值，以避免不必要的垃圾收集。

5.4.1.4　调整垃圾收集器

堆被分成年轻代和年老代。新对象分配在年轻代，如果该对象持续足够长时间的活跃（被使用），则移动到年老代。当一个被填满时，垃圾收集器开始回收工作，在此期间，进程中的所有其他线程都被暂停。年轻代很快被回收，因为暂停时间与活跃对象集相关，而与年轻代的大小无关。在年老代中，暂停时间大致与堆的大小相关。基于此，堆大小需合理分

配，使得事务和查询状态从不使它到年老代。堆大小参数设置是：neo4j-wrapper.conf 文件中的 dbms.memory.heap.max_size（以 MB 为单位）为最大堆大小，dbms.memory.heap.initial_size 为堆的初始大小，或者使用-Xms???m 开关参数指定。如果未指定，则由 JVM 自行选择。JVM 将根据需要自动增长堆，直到最大值。堆的增长需要一个完全的垃圾收集周期。建议将初始堆大小和最大堆大小设置为相同的值。这样，可以避免当垃圾收集器增大堆时发生的暂停。

新老两代堆之间的大小比例由-XX:NewRatio=N 开关参数控制，默认情况下，N 通常在 2 和 8 之间。比率为 2 意味着年老代大小除以年轻代大小等于 2，即：堆内存的三分之二将专用于年老代。同理，比率为 3 则将四分之三的堆分配给年老代，比率为 1 则两代的大小相同。比率 1 是相当激进的，不过适合事务更改了大量数据的场景。如果运行需要保留大量数据的 Cypher 查询，例如在排序大型结果集时，设置大的年轻代大小也很重要。

如果年轻代太小，短暂的对象可能会被过早地移动到年老代，这被称为提前升级，将增加年老代垃圾收集周期的频率，从而降低数据库性能。如果年轻代太大，垃圾收集器可能会决定年老代没有足够的空间来容纳所有希望从新的到旧的对象。这将年轻代垃圾收集周期变成年老代垃圾收集周期，再次减慢数据库的速度。运行更多并发线程意味着在给定的时间跨度内可能发生更多的分配，从而额外增加年轻代的压力。

JVM 中的压缩普通对象指针（Ordinary Object Pointer，OOP）功能仅支持 32 位。该功能可节省大量内存，但大于 32GB 的堆无法使用。因此，堆大小超过 32GB 收益可能很小，甚至带来副作用，除非增加非常得多（64GB 或以上）。

Neo4j 有一些长寿命的对象，它们留在年老代，有利于 Java 进程的生命周期。为了有效地处理它们，并且不会对垃圾收集暂停时间产生负面影响，我们建议使用并发垃圾收集器。如何调整特定的垃圾收集算法取决于 JVM 版本和工作负载。为获取有效的垃圾收集参数设置，建议在现有负载下测试几天或几周，堆碎片等问题可能需要更长的时间才能显现。为了获得良好的性能，这些是首先要研究的事情：

- 确保 JVM 不需要花太多时间来执行垃圾收集。目标是有一个足够大的堆，以确保重/峰值负载不会导致所谓的 GC（Garbage Collection）垃圾。当 GC 垃圾发生时，性能可能下降两个数量级。而堆太大也可能会降低性能，因此需要尝试一些不同的堆大小。
- 使用并发垃圾收集器。大多数情况下使用-XX:+UseG1GC 开关参数系统可良好工作。Neo4j JVM 需要足够的堆内存用于事务状态和查询处理，以及用于垃圾收集。因为堆内存需求是依赖于工作负载的，所以通常可配置 1GB 到高达 32GB。
- 使用-server 开关参数和一个大小合适的堆来启动 JVM。专用服务器上的操作系统通常可以使用 1GB 到 2GB 的内存。机器的物理内存越多，操作系统需要的内存越多。

neo4j-wrapper.conf 文件中涉及可配置 JVM 的属性如表 5-39 所示。

表 5-39　JVM 相关配置参数

属性名	说明
dbms.memory.heap.initial_size	初始堆大小（单位 MB）
dbms.memory.heap.max_size	最大堆大小（单位 MB）
dbms.jvm.additional	附加文字 JVM 参数

5.4.2　事务日志

本节介绍 Neo4j 日志的保留和轮换策略及其配置。事务日志记录了数据库的所有操作，是数据库恢复的重要凭证。事务日志支持增量备份与集群操作，至少需保留最新的非空事务日志。默认情况下，当日志文件大小超过 250MB 时，Neo4j 系统将自行切换日志文件，可通过 dbms.tx_log.rotation.size 参数进行手动配置。另外一个参数可控制保留的事务日志的数量，即：dbms.tx_log.rotation.retention_policy，其配置的格式是：

```
dbms.tx_log.rotation.retention_policy=<true/false>
dbms.tx_log.rotation.retention_policy=<amount> <type>
```

例如：

```
# Will keep logical logs indefinitely
dbms.tx_log.rotation.retention_policy=true
# Will keep only the most recent non-empty log
dbms.tx_log.rotation.retention_policy=false
# Will keep logical logs which contains any transaction committed within 30
days
dbms.tx_log.rotation.retention_policy=30 days
# Will keep logical logs which contains any of the most recent 500 000
transactions
dbms.tx_log.rotation.retention_policy=500k txs
```

全部参数如表 5-40 所示。

表 5-40　事务日志配置参数

类型	描述	示例
files	要保留的最近的逻辑日志文件的数量	10 files
size	允许日志文件占用的最大磁盘大小	300MB size 或 1GB size
txs	要保留的最新事务数	250k txs 或 5M txs
hours	保留包含从当前时间起 N 小时内提交的任何事务的日志	10 hours
days	保留包含从当前时间起 N 天内提交的任何事务的日志	50 days

5.4.3　压缩存储

本节介绍 Neo4j 属性值压缩和磁盘使用情况。在很多情况下，Neo4j 可以压缩和内联存储属性值，例如短数组和字符串，其目的是节省磁盘空间和提高 I/O 操作性能。

5.4.3.1　短数组的压缩存储

Neo4j 可用压缩的方式存储原始数组，采用“位剃削”算法，以减少存储数组中成员的

位数。尤其是：

- 对于数组的每个成员，确定最左边设置位的位置。
- 确定数组中所有成员中最大的位置。
- 它将所有成员减少到该位数。
- 存储这些值，前缀为一个小标题。

这意味着，当在数组中包括单个负值时，将使用原始字节大小来存储。在以下情况下，可能会将结果内联到属性记录中：

- 压缩后小于 24 个字节。
- 数组成员不到 64 个。

例如，数组 long[] {0L, 1L, 2L, 4L}内联，因为最大项（4）需要 3 位来存储，所以整个数组需 4×3=12 位存储。然而，数组 long[] {-1L, 1L, 2L, 4L}将需要整个 64 位用于-1 这个元素，因此它需要 64×4=256 字节，并采用动态存储。

5.4.3.2 短字符串的压缩存储

Neo4j 将对字符串按短字符串来分类、管理和处理。因此需进行显式存储，将其内联在属性记录中，即不支持可变长度字符串存储，以减少磁盘空间。此外，字符串记录无须存储属性，可在单个查找中进行读写操作，从而提高性能并减少磁盘空间。

短字符串种类有：

- 数字：由数字 0..9 和标点符号（空格、句点、破折号、加号、逗号和撇号）组成。
- 日期：由数字 0..9 和标点符号（空格、破折号、冒号、斜杠、加号和逗号）组成。
- 十六进制（小写）：由数字 0..9 和小写字母 a~f 组成。
- 十六进制（大写）：由数字 0..9 和大写字母 A~F 组成。
- 大写：由大写字母 A~Z，以及标点符号（空格、下划线、句点、破折号、冒号和斜杠）组成。
- 小写：如上，但字母为小写 a~z。
- 电子邮件：包含小写字母 a~z 和标点（逗号、下划线、句点、破折号、加号和@符号）。
- URI：由小写字母 a~z、数字 0~9 和大多数标点符号组成。
- 字母数字：由大写字母 A~Z、小写字母 a~z、数字 0~9 和标点符号（空格和下划线）组成。
- 阿尔法符号：由大写字母 A~Z、小写字母 a~z 和标点符号（空格、下划线、句点、破折号、冒号、斜杠、加号、逗号、撇号、符号、管道和分号）组成。
- 欧洲字符：包括大多数重音的欧洲字符和数字加标点符号空间（破折号、下划线和句号），如 latin1，但标点符号较少。
- Latin1。
- UTF-8。

除了字符串的内容之外，字符数还决定了字符串是否可以内联。每类都有字符数限制，限制详情请见表 5-41。

表 5-41　字符数限制表

字符串分类	字符数限制
数字、日期和十六进制	54
大写、小写和电子邮件	43
URI、字母数字和字母	36
欧洲字符	31
Latin1	27
UTF-8	14

这意味着最大的可内联字符串的长度为 54 个字符，必须为数字类。所有长度为 14 或更小的字符串将始终内联。还要注意，上述限制是针对参数 PropertyRecord 默认为 41 字节，如果通过修改源代码重新编译来更改该参数，则必须重新计算上述内容。

5.4.4　Linux 文件系统调优

本节介绍 Neo4j 输入/输出行为以及如何优化磁盘操作。数据库在查询数据时通常会产生许多少量、随机的读操作，而在提交更改时经常会产生少量的顺序写操作。默认情况下，大多数 Linux 发行版使用完全公平排队（Completely Fair Queuing，CFQ）算法来调度 IO 请求，该算法在吞吐量和延迟之间作了较好的平衡。然而，期限调度器（Deadline Scheduler）更适合于数据库的特定 IO 工作负载情形。期限调度器优先读操作请求，并尽快处理，以利于减少读操作的等待时间，而增加了写操作的等待时间。由于写操作通常是顺序的，增加其在 IO 队列中的延迟有益于合并重叠或相邻的写入请求，可有效地减少发送到驱动器的写入次数。Linux 系统中 IO 调度程序可在运行时修改驱动器（sda）：

```
echo 'deadline' > /sys/block/sda/queue/scheduler
$ cat            /sys/block/sda/queue/scheduler
noop [deadline] cfq
```

另一个推荐做法是禁用文件和目录的访问时间更新，文件系统将不对此元数据进行写操作更新，从而提高写入性能。可在 fstab 文件中设置 noatime、nodiratime 挂载选项来实现。

5.4.5　磁盘、内存及相关提示

本节指出在运行 Neo4j 时，磁盘和内存的注意事项。与任何持久性解决方案一样，性能取决于使用的持久性介质，更好的磁盘等于更好的性能。如果有多个磁盘或持久性介质可用，最好在这些磁盘之间划分存储文件和事务日志。将存储文件保存在具有低寻道时间的磁盘上将大大提高读取操作的性能。市面上，常见的机械硬盘的平均寻道时间约为 5 毫秒。当分配给页缓存的内存量太小时，这可能导致查询或遍历非常慢。而 SATA 接口的固态硬盘的平均寻道时间将小于 100 微秒，这意味着至少快 50 倍，但仍然是访问内存的几十或几百倍。

为了减少磁盘访问可加大内存容量。对普通的机械硬盘而言，使用 2~3GB 内存可处理几千万个图元（节点、关系和属性），8~16GB 内存的服务器可处理数亿个图元，16~64GB 内

存的服务器可处理数十亿个图元。而采用固态硬盘，则能够在更少的内存情况下，处理更多的图元。

可使用 dstat 或 vmstat 等工具来收集应用程序的运行信息。如果交换或分页数很高，则表明 Lucene 索引不能全部加载到内存，将导致索引查找延迟增大。在 Neo4j 启动时，页面缓存为空，需要预热，但特别是对于大型存储，这可能需要更长的预热时间，从驱动器读取很多块，并且 IO 等待时间很长，这很正常。Neo4j 还在后台刷新其页缓存，因此很少看到在稳态期间一个连续的块被写入驱动器。然而，这种后台刷新仅产生少量的 IO 等待。如果在稳态期间 IO 等待时间较长，则可能是 Neo4j 对驱动器的随机 IO 性能造成的。运行 Neo4j 最好的驱动器是快速固态硬盘，它可进行大量随机的读写操作（Input/Output Operations Per Second，IOPS）。

5.5 Neo4j 数据库的备份与恢复

本节主要介绍 Neo4j 数据库的备份与恢复，涵盖因果集群（Causal Cluster）、高可用集群（Highly Available Cluster）和单实例（Single Instance）等各种配置方式的 Neo4j 数据库。本节功能仅适用于 Neo4j 企业版。

5.5.1 备份简介

将 Neo4j 数据库备份到远程或离线存储是一项基本操作，Neo4j 支持全量备份和增量备份。对于三种配置方式（单实例、高可用集群和因果集群）的 Neo4j 数据库，其备份过程是相同的。只是对于因果集群，需注意区分核心副本角色（Core Replica Roles）和只读副本角色（Read Replica Roles），以确定哪些最适合作为备份服务器，详情请参见“5.5.2.3 备份因果集群”小节，同样，恢复 Neo4j 因果集群也有点不同，详情请参见“5.5.3.3 恢复因果集群”小节。备份使用 neo4j-backup 工具在网络上运行，即可从 Neo4j 服务器备份到本地副本。

5.5.1.1 启用备份

必须配置两个参数才能执行备份。

- dbms.backup.enabled=true：将启用备份，这是默认值。
- dbms.backup.address=<主机名或 IP 地址>:6362：配置备份服务侦听的接口和端口，默认值为环回接口和 6362 端口，设置 dbms.backup.address=0.0.0.0:6362 则表示侦听所有地址。

5.5.1.2 存储注意事项

备份重要的是数据与生产系统没有共同依赖关系，并分开存储。建议将备份保存在集群服务器之外的存储上，比如：优选云，同一云中的不同可用性区域或单独的云等。由于备份保存很长时间，归档存储的长期性应被视为备份计划的一部分。有关可用配置选项的详情请

仅供非商业用途或交流学习使用

参阅官方参考资料“配置设置参考[20]”，也可参见 Neo4j 数据库安装目录下的 neo4j.conf 文件，neo4j.conf 配置参见“附录 A　Neo4j 配置设置”。

5.5.2　执行备份

本小节介绍如何执行 Neo4j 数据库的备份。

5.5.2.1　备份命令

neo4j-admin 工具位于 bin 目录下，使用 backup 参数运行它，以便对正在运行的数据库执行联机备份。语法为：neo4j-admin backup --backup-dir=<backup-path> --name=<graph.db-backup> [--from=<address>] [--fallback-to-full[=<true|false>]] [--check-consistency[=<true|false>]] [--cc-report-dir=<directory>] [--additional-config=<config-file-path>] [--timeout=<timeout>]。备份命令参数说明如表 5-42 所示。

表 5-42　备份命令参数说明

参数名	默认值	说明
--backup-dir		存放备份的目录
--name		备份名。如果具有此名的备份已存在，将尝试增量备份
--from	localhost:6362	Neo4j 的主机和端口
--fallback-to-full	true	如果增量备份失败，备份过程将旧备份改名为<name>.err.<N>，并由增量备份方式改为全量备份
--check-consistency	true	是否进行一致性检查
--cc-report-dir	.	写入一致性报告的目录
--additional-config		提供附加配置的配置文件
--timeout	20m	以<time>[ms|s|m|h]的形式来设置超时，其中默认单位为秒

下面命令为：创建一个空目录并运行 backup 命令，执行全量备份。

```
$neo4j-home> mkdir /mnt/backup
$neo4j-home> bin/neo4j-admin backup --from=192.168.1.34 --backup-
dir=/mnt/backup --name=graph.db-backup
Doing full backup...
2017-02-01 14:09:09.510+0000 INFO  [o.n.c.s.StoreCopyClient] Copying
neostore.nodestore.db.labels
2017-02-01 14:09:09.537+0000 INFO  [o.n.c.s.StoreCopyClient] Copied
neostore.nodestore.db.labels 8.00 kB
2017-02-01 14:09:09.538+0000 INFO  [o.n.c.s.StoreCopyClient] Copying
neostore.nodestore.db
2017-02-01 14:09:09.540+0000 INFO  [o.n.c.s.StoreCopyClient] Copied
neostore.nodestore.db 16.00 kB
...
...
...
```

执行完上述操作后，将在/mnt/backup 目录下看到一个名为 graph-db.backup 的备份。

[20] https://neo4j.com/docs/operations-manual/current/reference/configuration-settings/

仅供非商业用途或交流学习使用

5.5.2.2 增量备份

增量备份需指定现有备份目录并且有自上次备份以来的事务日志。备份工具将上次之后的任何操作进行备份，其结果将是与当前服务器状态一致的更新备份。增量备份可能会失败，主要原因有：现有目录不包含有效的备份，并且参数 fallback-to-full=false；所需的事务日志已删除且参数 fallback-to-full=false。因此，设置参数 fallback-to-full=true 是一种安全措施，它能在增量备份失败的情况下转为全量备份，以防止无法执行增量备份。在指定先前备份的位置后，执行增量备份的命令如下：

```
$neo4j-home> bin/neo4j-admin backup --from=192.168.1.34 --backup-
dir=/mnt/backup --name=graph.db-backup --fallback-to-full=true --check-
consistency=true
Destination is not empty, doing incremental backup...
Backup complete.
```



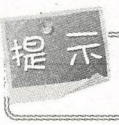
请注意，复制未完成的事务时，服务器需要访问事务日志。这些日志由 Neo4j 维护，并在一段时间后基于参数 dbms.tx_log.rotation.retention_policy 自动删除。因此，在设计备份策略时，重要的是配置 dbms.tx_log.rotation.retention_policy，以便事务日志保存在增量备份之间。

5.5.2.3 备份因果集群

在 Neo4j 因果集群中，核心服务器（Core Servers）和只读副本（Read Replicas）都支持备份协议，都可用于集群备份。以下是在确定要使用备份策略前需考虑的注意事项。

1. 只读副本备份

通常，更倾向于选择只读副本作为备份，因为只读副本的数量在因果集群部署中远远多于核心服务器。但是，只读副本是从核心服务器上异步复制的，因此存在事务执行上的差异，甚至其内容已相当陈旧。幸运的是，可以检查在任何服务器上处理的最后一个事务 ID，这样就可以验证它是否足够接近由核心服务器处理的最新事务 ID。如果是则可以安全地读取副本备份，以保证它是核心服务器上最新的。



Neo4j 数据库中的事务 ID 是严格递增的整数值。因此，事务 ID 值越大则表明该事务发起时间越新。

Neo4j 服务器可通过 JMX 或 Neo4j 浏览器查看最后一个事务 ID。在 Neo4j 浏览器中查看最新处理的事务 ID（及其他指标），可在提示符下输入：sysinfo。

2. 核心服务器备份

在只有核心服务器的集群（Core-only Cluster）中，就没有大量的只读副本来分担工作负载。因此，在选择哪台核心服务器进行备份操作时，需要考虑服务器方面的因素有：物理接

仅供非商业用途或交流学习使用

近性、带宽、性能及活跃度等。

一般情况下，即使集群在发生大量备份时，仍能正常工作，但是，备份操作将对备份服务器增加额外的 IO 负担，可能会影响其性能。比较保守的做法是：将备份服务器当作不可用实例，假定其性能低于集群中的其他实例。在这种情况下，建议在集群中有足够的冗余，以便一个较慢的服务器不会减少屏蔽故障的能力。可以在集群规划时，就考虑这种保守的策略。等式 M = 2F + 1，M 是集群中需要容忍 F 个故障成员之间的关系。为了容忍在备份期间在集群中一个较慢的机器的可能性，可增大 F 值，比如，最初设想三个核心服务器的集群容忍一个故障，则可以将其增加到 5 个以确保相应的安全级别。

5.5.3　恢复备份

本节介绍如何从 Neo4j 数据库的备份进行恢复。

5.5.3.1　恢复单实例数据库

Neo4j 备份是全功能的数据库备份。恢复备份，必须关闭数据库，使用 neo4j-admin 工具的 restore 参数恢复备份。例如，从位于/mnt/backup/neo4j-backup 中的备份恢复数据库 graph.db 的操作命令如下：

```
neo4j-home> bin/neo4j stop
neo4j-home> bin/neo4j-admin restore --from=/mnt/backup/neo4j-backup --
database=graph.db --force
neo4j-home> bin/neo4j start
```

5.5.3.2　恢复高可用集群

要从高可用集群环境中的备份进行恢复，请按照下列步骤进行：

- 关闭集群中的所有数据库实例。
- 恢复每个实例上的备份。
- 启动数据库实例。

5.5.3.3　恢复因果集群

恢复 Neo4j 因果集群的步骤与初始创建因果集群的步骤相同。除非是一个全新的 Neo4j 因果集群数据库，否则都需要从已有的单实例 Neo4j 数据库或高可用 Neo4j 集群导入因果集群中，即：创建或恢复因果集群。假设已经从单实例 Neo4j 数据库或高可用 Neo4j 集群中创建了一个备份，并且存储在 seed-dir/目录下。恢复步骤为：先创建一个新的 Neo4j 核心集群，不要启动组成集群的实例，而进行数据库内容的恢复。使用 neo4j-admin 工具的 restore 命令将存储版本从 seed-dir/目录恢复到因果集群中每个核心实例（Core instances）上。命令如下：

```
neo4j-01$ ./bin/neo4j-admin restore --from=seed-dir/ --database=graph.db
neo4j-02$ ./bin/neo4j-admin restore --from=seed-dir/ --database=graph.db
neo4j-03$ ./bin/neo4j-admin restore --from=seed-dir/ --database=graph.db
```

仅供非商业用途或交流学习使用



上面的命令行假定数据库　名称默认为 graph.db。如果已配置为其他数据库名称，请将此命令行参数做相应的更改。

至此，因果集群中的每一个核心实例均包含所指定的数据库文件，其内部可自动形成一个集群，只需继续启动因果集群的所有实例即可。命令如下：

```
neo4j-01$ ./bin/neo4j start
neo4j-02$ ./bin/neo4j start
neo4j-03$ ./bin/neo4j start
```

在恢复时，整个因果集群的可用性是非常高的，因为集群在上线之前就启动内部处理，同时还可以使用联机只读副本提供查询服务。



如果因果集群未按预期形式启动，其日志中将包含足够的信息供操作员来定位问题。

如果要导出因果集群的数据内容以创建单实例或高可用 Neo4j 数据库部署，首先需要在因果集群上执行备份，再将此备份文件作为其他部署形式的数据源文件进行恢复。

5.6 Neo4j 数据库管理相关工具

本节主要介绍与 Neo4j 数据库管理密切相关的几个常用工具，包括：Neo4j 数据导入工具、Cypher Shell、Neo4j 数据转储和加载，以及一致性检查工具。

5.6.1 导入工具

导入工具用于将保存为 CSV 文件的数据导入到新的 Neo4j 数据库。重点介绍工具的使用和输入数据的格式化。采用 CSV 文件加载数据时，每个节点必须有唯一标识符，称为节点标识符（Node Identifier），以便在同一进程中的节点之间创建关系，关系是通过节点标识符之间的连接来创建。在下面的示例中，节点标识符作为属性存储在节点上，节点标识符可用于交叉引用，以利于图的回溯，但不强制性使用。如果不希望在导入后还保留标识符，不指定属性的 ID 字段即可。

5.6.1.1 简单导入示例

以电影、演员和角色图数据集为例，先将 path_to_target_directory 修改为数据库文件目录，默认安装下 path_to_target_directory 指向<neo4j-home>/data/databases/graph.db 文件，其中电影节点文件为 movies.csv，每部电影都有一个编号 id，便于外部数据源的引用，另外每部电影都有电影名和年份属性，并为每个节点添加了 Movie 和 Sequel 标签。该文件内容示例如下：

仅供非商业用途或交流学习使用


```
movieId:ID,title,year:int,:LABEL
tt0133093,"The Matrix",1999,Movie
tt0234215,"The Matrix Reloaded",2003,Movie;Sequel
tt0242653,"The Matrix Revolutions",2003,Movie;Sequel
```

演员节点文件为 actors.csv，每个演员有两个属性：编号（姓名的缩写）、姓名，并带有 Actor 属性，该文件内容示例如下：

```
personId:ID,name,:LABEL
keanu,"Keanu Reeves",Actor
laurence,"Laurence Fishburne",Actor
carrieanne,"Carrie-Anne Moss",Actor
```

接下来，角色文件为 roles.csv，保存演员与电影之间的关系，START_ID 为演员节点中的编号，END_ID 为电影中的编号，role 字段为该演员在这部电影中所扮演的角色名，TYPE 字段为关系类型（在本例中为 ACTED_IN）。该文件内容示例如下所示：

```
:START_ID,role,:END_ID,:TYPE
keanu,"Neo",tt0133093,ACTED_IN
keanu,"Neo",tt0234215,ACTED_IN
keanu,"Neo",tt0242653,ACTED_IN
laurence,"Morpheus",tt0133093,ACTED_IN
laurence,"Morpheus",tt0234215,ACTED_IN
laurence,"Morpheus",tt0242653,ACTED_IN
carrieanne,"Trinity",tt0133093,ACTED_IN
carrieanne,"Trinity",tt0234215,ACTED_IN
carrieanne,"Trinity",tt0242653,ACTED_IN
```

接下来调用数据导入 neo4j-import 命令如下所示：

```
neo4j_home$ ./bin/neo4j-import --into path_to_target_directory --nodes
movies.csv --nodes actors.csv --relationships roles.csv
```

完成后则可以启动数据库，命令如下所示：

```
neo4j_home$ ./bin/neo4j start
```

在创建输入文件时需要注意如下几点：

- 默认情况下，字段以逗号分隔，但可以指定其他分隔符。
- 所有文件必须使用相同的分隔符。
- 节点和关系可以保存在多个数据源。
- 数据源可来源于多个文件。
- 提供数据字段信息的标题必须位于每个数据源的第一行。
- 在标题中没有相应信息的字段将不会被导入。
- 采用 UTF-8 编码。



导入时不需要创建索引，导入完成后再添加。如果无法使用此工具进行数据导入，且加载的 CSV 文件为中小型，可以使用 LOAD CSV 方式。

仅供非商业用途或交流学习使用

5.6.1.2 CSV 文件头格式

每个数据源的标题行负责解释文件中的字段，标题行与余下数据行具有相同的定界符。每个字段的格式为：<name>:<field_type>，<name>为属性值，<field_type>用于节点和关系：

（1）属性值（Property value）

数据类型可以是：int、long、float、double、boolean、byte、short、char、string。如果未指定数据类型，则默认为 string 类型。数组类型在上述类型后面加上[]即可。默认情况下，数组值由分号（;）分隔，也可以使用--array-delimiter 指定为其他分隔符。

（2）节点（Node）

节点须有 ID 和 LABEL 字段，每个节点在导入前必须指定唯一编号（ID），以利于在创建关系时能正确查找到相应的节点。ID 必须唯一，即使某节点具有不同的标签。LABEL 字段用于读取一个或多个标签，与数组类似，多个标签由分号（;）分隔，或由--array-delimiter 来指定分隔符。

（3）关系（Relationship）

对于关系数据源，有三个必填字段：TYPE、START_ID 和 END_ID。TYPE 为关系类型，START_ID 为关系起始节点编号，END_ID 为关系终止节点编号。

导入工具默认假定节点标识符在节点之间是唯一的。如果不唯一，还可以定义一个编号空间（ID space），编号空间由节点文件的 ID 字段来定义。例如，要指定 Person 的编号空间，只需在 people 节点文件中使用字段类型 ID(Person)，在关系文件（即 START_ID(Person) 或 END_ID(Person)）中引用该编号空间即可。

5.6.1.3 命令行运行导入工具

在 UNIX/Linux/OSX 操作系统中，导入工具命令名为 neo4j-import。与安装配置相关，导入工具可能在任意路径下调用，可能仅在安装目录中执行。而 Windows 系统则在安装目录下执行 bin\neo4j-import 命令即可。该命令有如下参数：

- --into <store-dir>：要导入到的数据库目录，该目录下不能包含已有数据库。
- --nodes[:Label1:Label2] "<file1>,<file2>,...": 节点 CSV 标题和数据，如果带有多个文件将在逻辑上视为一个大文件。第一行必须为标题，每个数据源都有自己的标题。注意，一组文件必须用引号括起来。
- --relationships[:RELATIONSHIP_TYPE] "<file1>,<file2>,...": 关系 CSV 标题和数据。同样，多个文件将在逻辑上视为一个大文件，第一行必须为标题，每个数据源都有自己的标题，一组文件必须用引号括起来。
- --delimiter <delimiter-character>: CSV 文件中数据分隔符或 TAB 制表符，默认为逗号。
- --array-delimiter <array-delimiter-character>: CSV 文件中数组元素之间的分隔符或 TAB 制表符，默认为分号。
- --quote <quotation-character>: 设定 CSV 数据中值的引用字符，默认为引号。引号里面的引号需要转义，比如："""Go away"", he said."和"\"Go away\", he said."两种方式

仅供非商业用途或交流学习使用

系统都支持。如果设置'为引用字符，则上例需要改写为："'Go away", he said.'。

- --multiline-fields <true/false>：输入字段是否可以跨行，即是否包含换行符，默认不包含，即值为：false。
- --input-encoding <character set>：输入数据的编码字符集，必须是 JVM 中的可用字符集，可通过 Charset # availableCharsets()方法查询获取；如果未指定编码，则使用 JVM 的默认字符集。
- --ignore-empty-strings <true/false>：空字符串字段是否被忽略，默认不忽略，值为：false。
- --id-type <id-type>：指定节点/关系中编号字段的数据类型，可以为：STRING、INTEGER、ACTUAL。STRING 为字符串，INTEGER 为整数值，ACTUAL 以实际节点标识为其类型。默认值为 STRING。
- --processors <max processor count>：（高级选项）指定导入工具可以使用的最多处理器数量，默认为 JVM 所检测的可用处理器数。为了获得最佳性能，此值不应大于可用处理器的数量。
- --stacktrace <true/false>：是否启用打印错误堆栈跟踪信息。
- --bad-tolerance <max number of bad entries>：容忍导入错误的数据条数，输入数据中的格式错误被视为错误，默认值为 1000。
- --skip-bad-relationships <true/false>：导入是否跳过缺失节点编号的关系，即关系中未指定开始或结束节点编号，默认跳过，值为 true。
- --skip-duplicate-nodes <true/false>：是否跳过导入相同编号的节点。默认值：false。
- --ignore-extra-columns <true/false>：是否忽略标题中未指定的额外数据列，默认值：false。
- --db-config <path/to/neo4j.conf>：此为高级选项，指定数据库特定配置的文件路径，与导入工具相关的配置参数有三个，分别为：dbms.relationship_grouping_threshold、unsupported.dbms.block_size.strings、unsupported.dbms.block_size.array_properties。

调用导入工具在某些情况下，可能引发意外错误，可以带上--stacktrace 参数打印错误信息，以利于开发和调试。在导入过程中，控制台上还将输出相关的统计数据，输出内容可分成几个部分，输出越宽，则导入消耗的时间越多，最宽的部分可认为是瓶颈，用*标记。如果有双行，而不是单行，则表明多线程运行。最右边显示一个数字，指出该阶段处理的实体（节点或关系）数量。例如：

```
[*>:20,25 MB/s----------|PREPARE(3)=========|RELATIONSHIP(2)=========] 16M
```

上例中“>”表示正在读取解析数据，速度为 2025 MB/s；PREPARE 为准备数据；RELATIONSHIP 为创建实际关系记录；v 表示正在写入的要存储的关系，该步骤在这个例子中没有显示，是因为这一步所花时间相对于其他步骤而言要少得多。通过输出的信息可以改善数据库系统的性能，上例中，瓶颈是在数据读取部分（用>标记处），这可能表明：磁盘速度较慢，或系统在导入数据的同时还在进行读写操作。

仅供非商业用途或交流学习使用

5.6.2 Cypher Shell

Cypher Shell 是 Neo4j 数据库的一个命令行工具，可以用于数据库连接、调用 Cypher 语句进行数据查询，或定义相关模式和执行管理任务。Cypher Shell 采用显式事务方式，允许将多个操作分组一并执行或回滚，通信方式采用加密的二进制 Bolt 协议。Cypher Shell 位于 bin 目录中，运行时可带上一组参数，第一次运行 Cypher Shell 时，系统将提示您输入相关的安全信息，按照提示输入即可。其语法为：cypher-shell [-h] [-a ADDRESS] [-u USERNAME] [-p PASSWORD] [--encryption {true,false}] [--format {verbose,plain}] [--debug] [--fail-fast | --fail-at-end] [cypher]。各参数的具体说明如表 5-43 所示。

表 5-43 Cypher Shell 参数说明

参数类型	参数名	描述
位置参数	cypher	cypher 执行后退出的可选字符串
可选参数	-h, --help	显示帮助消息并退出
	--fail-fast	文件读取时，遇到第一个错误就退出并报告失败（这是默认行为）
	--fail-at-end	文件读取时，在输入结束时退出并报告错误
	--format {verbose,plain}	指定输出格式，verbose 显示统计信息，plain 仅显示数据，默认值为：verbose
	--debug	打印调试信息（默认值：false）
连接参数	-a ADDRESS, --address	要连接的 IP 地址和端口（默认值：localhost:7687）
	-u USERNAME, --username	连接的用户名，也可以使用环境变量 NEO4J_USERNAME 指定（默认为空）
	-p PASSWORD, --password	连接的密码，也可以使用环境变量 NEO4J_PASSWORD 指定（默认为空）
	--encryption {true,false}	连接是否加密，必须与 Neo4j 的配置一致（默认值：true）

下面给出几个使用 Cypher Shell 的示例：

（1）使用用户名和密码调用 Cypher Shell。

```
$neo4j-home> bin/cypher-shell -u johndoe -p secret
Connected to Neo4j at bolt://localhost:7687 as user neo4j.
Type :help for a list of available commands or :exit to exit the shell.
Note that Cypher queries must end with a semicolon.
neo4j>
```

（2）从 Cypher Shell 中调用帮助。

```
neo4j> :help
Available commands:
  :begin    Open a transaction
  :commit   Commit the currently open transaction
  :exit     Exit the logger
  :help     Show this help message
  :history  Print a list of the last commands executed
  :param    Set the value of a query parameter
  :params   Prints all currently set query parameters and their values
  :rollback Rollback the currently open transaction
For help on a specific command type:
  :help command
```

仅供非商业用途或交流学习使用

（3）在 Cypher Shell 中执行查询。

```
neo4j > MATCH (n) RETURN n;
n
(:Person {name: "Bruce Wayne", alias: "Batman"})
(:Person {name: "Selina Kyle", alias: ["Catwoman", "The Cat"]})
```

（4）从命令行使用 Cypher 脚本调用 Cypher Shell。下面是一个名为 examples.cypher 文件中的内容：

```
MATCH (n) RETURN n;
MATCH (batman:Person {name: 'Bruce Wayne'}) RETURN batman;
```

从命令行调用 examples.cypher 脚本，可以使用--format plain 标志来限制输出：

```
$neo4j-home> cat examples.cypher | bin/cypher-shell -u neo4j -p maria --
format plain
n
(:Person {name: "Bruce Wayne", alias: "Batman"})
(:Person {name: "Selina Kyle", alias: ["Catwoman", "The Cat"]})
batman
(:Person {name: "Bruce Wayne", alias: "Batman"})
```

Cypher Shell 支持基于参数的查询，通常在编写脚本时使用。比如：使用':param'关键字将'thisAlias'设置为'Robin'，这样就可以使用':params'关键字来检验参数是否设置正确。

```
neo4j> :param thisAlias 'Robin'
neo4j> :params
thisAlias: Robin
```

现在在 Cypher 查询中就可以使用参数'thisAlias'，下面示例可以进行结果验证。

```
neo4j> CREATE (:Person {name : 'Dick Grayson', alias : {thisAlias} });
Added 1 nodes, Set 2 properties, Added 1 labels
neo4j> MATCH (n) RETURN n;
n
(:Person {name: "Bruce Wayne", alias: "Batman"})
(:Person {name: "Selina Kyle", alias: ["Catwoman", "The Cat"]})
(:Person {name: "Dick Grayson", alias: "Robin"})
```

Cypher Shell 支持显式事务，使用关键字:begin、:commit 和:rollback 来控制事务的状态。例如在第一个 Cypher Shell 会话中启动事务：

```
neo4j> MATCH (n) RETURN n;
n
(:Person {name: "Bruce Wayne", alias: "Batman"})
(:Person {name: "Selina Kyle", alias: ["Catwoman", "The Cat"]})
(:Person {name: "Dick Grayson", alias: "Robin"})
neo4j> :begin
neo4j# CREATE (:Person {name : 'Edward Mygma', alias : 'The Riddler' });
Added 1 nodes, Set 2 properties, Added 1 labels
```

仅供非商业用途或交流学习使用

如果现在打开第二个 Cypher Shell 会话，将看到最近的 CREATE 语句没有更改：

```
neo4j> MATCH (n) RETURN n;
n
(:Person {name: "Bruce Wayne", alias: "Batman"})
(:Person {name: "Selina Kyle", alias: ["Catwoman", "The Cat"]})
(:Person {name: "Dick Grayson", alias: "Robin"})
```

回到第一个会话并提交事务：

```
neo4j# :commit
neo4j> MATCH (n) RETURN n;
n
(:Person {name: "Bruce Wayne", alias: "Batman"})
(:Person {name: "Selina Kyle", alias: ["Catwoman", "The Cat"]})
(:Person {name: "Dick Grayson", alias: "Robin"})
(:Person {name: "Edward Mygma", alias: "The Riddler"})
neo4j>
```

Cypher Shell 支持运行当前用户已授权的任何过程。例如，调用本地内置过程 dbms.security.showCurrentUser()。

```
neo4j> CALL dbms.security.showCurrentUser();
```

结果

```
username, roles, flags
"johndoe", ["admin"], []
neo4j> :exit
Exiting. Bye bye.
Bye!
```

5.6.3 Neo4j 数据转储和加载

Neo4j 数据转储和加载需要用到 neo4j-admin 命令，该命令格式为：

```
neo4j-admin dump --database=<database> --to=<destination-path>
neo4j-admin load --from=<archive-path> --database=<database> [--force]
```

通过使用上述命令可将数据库从一个环境移动到另一个环境，也可用于数据库的脱机备份。例如：将名为 graph.db 的数据库转储到名为/backups/graph.db/2016-10-02.dump 的文件中，转储文件的目标目录（在本例中为/backups/graph.db）在调用命令之前必须存在。

```
$neo4j-home> bin/neo4j-admin dump --database=graph.db --
to=/backups/graph.db/2016-10-02.dump
$neo4j-home> ls /backups/graph.db
$neo4j-home> 2016-10-02.dump
```

还可以将包含在文件/backups/graph.db/2016-10-02.dump 中的备份数据库加载到数据库 graph.db 中。但由于当前数据库正在运行，先将其关闭，再使用--force 选项来覆盖现有数据库中的数据。操作方法如下：

仅供非商业用途或交流学习使用


```
$neo4j-home> bin/neo4j stop
Stopping Neo4j.. stopped
$neo4j-home> bin/neo4j-admin load --from=/backups/graph.db/2016-10-02.dump -
-database=graph.db --force
```



虽然 Neo4j 数据库已经考虑到了不同历史版本中文件格式和数据内容的兼容，像复制和粘贴数据目录文件方式可以很方便地将数据在不同数据库系统之间进行传输，然而，数据库目录的结构和内容可能随时改变，从而难以保证复制粘贴数据文件方式的正常运行，建议采用 neo4j-admin dump 和 neo4j-admin load 命令这种更安全的全库导出、导入方式。

5.6.4　一致性检查工具

一致性检查可以使用 neo4j-admin 工具的 check-consistency 参数来检查数据库的一致性。该 neo4j-admin 工具位于 bin 目录中，语法调用为：neo4j-admin check-consistency --database=<database> [--report-dir=<directory>] [--additional-config=<file>] [--verbose]。命令中的参数详细介绍如表 5-44 所示。

表 5-44　一致性检查命令参数说明

参数名	描述
--database	指定需要进行一致性检查的数据库名称
--report-dir	写入报告的目录，默认为工作目录
--additional-config	附加配置文件，用于一致性检查的附加配置
--verbose	启用详细信息输出，包括存储信息和内存使用信息

一致性检查工具不能与当前正在使用的数据库一起使用，如果与正在运行的数据库一起使用，则自动停止并输出错误信息。如果一致性检查工具没有发现错误，则程序自动运行结束并不生成相应的报告；如果发现错误，程序将退出，退出代码为 1，并将错误信息写入一个格式为 inconsistencies-YYYY-MM-DD.HH24.MI.SS.report 的报告文件中，此文件的位置在当前工作目录下，或由参数 report-dir 指定。例如运行一致性检查工具显示如下：

```
$neo4j-home> bin/neo4j-admin check-consistency --database=graph.db
2016-09-30 14:00:47.287+0000 INFO  [o.n.k.i.s.f.RecordFormatSelector] Format
not configured. Selected format from the store:
RecordFormat:StandardV3_0[v0.A.7]
..................  10%
..................  20%
..................  30%
..................  40%
..................  50%
.............Checking node and relationship counts
..................  10%
..................  20%
..................  30%
..................  40%
..................  50%
```

仅供非商业用途或交流学习使用


```
..................  60%
..................  70%
..................  80%
..................  90%
.................. 100%
```

一致性检查工具可以调用由参数 additional-config 指定的配置文件中的其他配置选项，配置文件的格式与 neo4j.conf 格式相同，如表 5-45 所示。

表 5-45 一致性检查附加配置参数

参数名	默认值	描述
tools.consistency_checker.check_graph	true	在节点、关系、属性、类型和令牌之间执行检查
tools.consistency_checker.check_indexes	true	对索引执行检查。检查索引比检查本地存储更花时间，建议大图检查关闭此项
tools.consistency_checker.check_label_scan_store	true	对标签扫描存储执行检查。此项检查比本地存储检查更耗时，建议大图检查关闭此项
tools.consistency_checker.check_property_owners	false	对属性所有者执行可选检查。这可以检测理论上的不一致性，其中属性可以由多个实体所有。但是，检查在时间和内存开销较大，默认情况下不检查

例如创建文件 consistency-check.properties，其内容为：

```
tools.consistency_checker.check_graph=false
tools.consistency_checker.check_indexes=true
tools.consistency_checker.check_label_scan_store=true
tools.consistency_checker.check_property_owners=false
```

接下来，使用配置文件来运行一致性检查工具：

```
$neo4j-home> bin/neo4j-admin check-consistency --database=graph.db --
additional-config=consistency-check.properties
```



仅供非商业用途或交流学习使用

第 6 章

◂Neo4j 集群技术▸

本章包括以下内容：

- 因果集群
- 高可用性集群
- 集群的备份与恢复
- 单实例和高可用集群版本升级

集群技术是一项在较低成本下能有效提高系统整体性能、可靠性、灵活性和可扩展性，并广泛应用于生产系统的必备技术，同样，Neo4j 在其企业版中也提供了数据库集群功能，分为两种：高可用性集群（High Availability，HA）和因果集群（Causal Clustering）。高可用性集群功能最先在 Neo4j 1.2.M05 (2010-12-02)版本上新增，并经过后续版本的不断改进和完善；因果集群是在最新的 Neo4j 3.1.0 企业版中才增加的，它采用当前更先进的集群架构和安全架构，以满足企业更大规模生产与安全性需求。

本章主要介绍 Neo4j 的两种集群的概念及其配置，Neo4j 集群是为了满足企业的大规模生产环境和容错而设计的。特别是因果集群是 Neo4j 3.1.0 才具有的新特性，Neo4j 因果集群提供了业界最先进的 ACID 兼容模型，因果一致性可以支持实现更多种类的应用，其中很多是最终一致性模型无法做到的。学完本章后将可以部署自己的因果集群和 HA 集群，如果要涉及集群中更为深层次的内容，可查阅官网操作指南。

本章内容更适合需要在大规模环境下部署生产系统的企业或研发人员。

6.1 因果集群

因果集群技术基于 Raft 协议开发，Raft 协议是由斯坦福大学提出的一种更易于理解的一致性算法，其用意在于取代目前广为使用的 Paxos 算法，目前，在各种主流程序设计语言中均有其开源实现。Neo4j 因果集群可支持数据中心和云所需的大规模和多拓扑环境，其中内置了由 Neo4j Bolt 驱动处理的负载均衡，还支持同样由 Bolt 驱动管理的集群可感知会话，有助于为开发人员解决集群架构上的诸多问题。

因果集群的安全性改进还包括：多用户、基于角色的访问控制（提供四种预定义的全局图数据访问角色：读取者、发布者、架构者和管理者）、查询安全事件日志、列出并终止运行中的查询，以及细粒度的访问控制等特性。



仅供非商业用途或交流学习使用

6.1.1 初识因果集群

Neo4j 因果集群提供了两个主要的特点：

- 安全性：核心服务器（Core）为事务处理提供容错平台，仅有一部分核心服务器正常运行时，整个系统仍然是可用的。
- 可扩展性：只读副本（Read Replica）为图查询提供了一个大规模高可扩展的平台，可以在分布广泛的拓扑中执行非常大的图查询工作。

以上两个特点结合起来，允许最终用户系统拥有数据库完整的功能，并且在发生多个硬件和网络故障的情况下也可以进行正常的数据库读写操作。

在本节的余下部分中，我们将介绍因果集群在生产中如何工作，包括操作和应用两个方面。

6.1.2 操作视图

从操作的角度来看，可以认为集群包括两个主要角色，一个是核心服务器（Core），另一个是只读副本（Read Replica）。因果集群架构如图 6-1 所示。

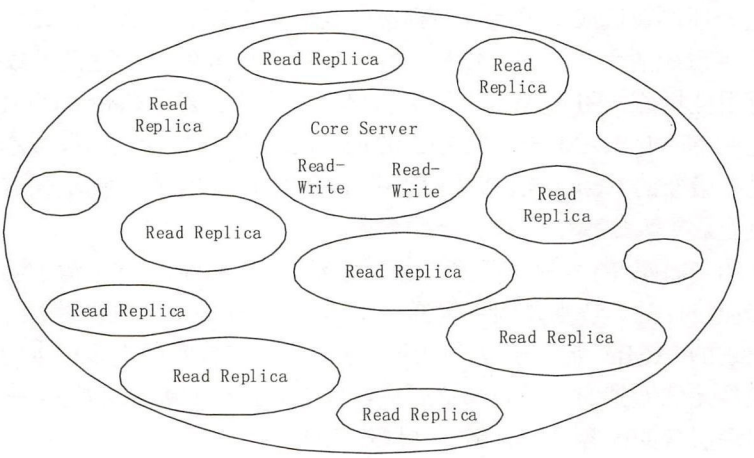


图 6-1 因果集群架构

这两个角色是任何生产部署中的基础，但是各自的规模不同，并且在管理整个集群的容错和可扩展性方面承担着不同的功能。

6.1.2.1 核心服务器

核心服务器的主要职责是保护数据，它是使用 Raft 协议复制所有事务来实现这个功能。Raft 协议能确保数据在最终用户应用程序提交事务之前是安全的。在实际中，这意味着一旦一个集群中的大多数（至少超过 50%，即：N/2+1，N 为集群中核心服务器数量）核心服务器已经接受该事务，就可以安全地提交给最终用户应用程序。

这种安全性要求将对写操作有延迟影响。隐含着：写操作将由最快的大多数核心服务器来进行确认，但随着集群中核心服务器数量的增长，这个数量有必要人工设定以降低写操作延迟。



仅供非商业用途或交流学习使用

实际上，这意味着在典型的核心服务器集群中只需相对较少的机器就可以为特定部署提供足够的容错能力。可以通过公式进行简单计算 M = 2F + 1，其中 M 是允许 F 个核心服务器发生故障所需的核心服务器总数。例如，为了容忍 2 个出现故障的核心服务器，我们需要部署一个至少拥有 5 个核心服务器的集群。

请注意，如果核心服务器集群发生故障而无法再处理写入操作，则该核心服务器将转化为只读状态，以确保整个集群的数据安全。

6.1.2.2　只读副本

只读副本的主要职责是扩展图操作的工作负载（比如：Cypher 查询、过程处理等）。只读副本就像是核心服务器中受保护数据的高速缓存，但它们不是简单的键值高速缓存。事实上，只读副本是能够完成任意（只读）图查询和过程处理的全功能的 Neo4j 数据库。

只读副本通过事务日志以异步的方式从核心服务器复制数据。只读副本会周期性地（通常在毫秒范围内）轮询核心服务器，以查找自上次轮询后处理的任何新事务，然后核心服务器将这些新事务发送到只读副本。大量的只读副本可以从相对较少的核心服务器复制数据，从而确保大量的图查询工作负载得以分摊。

与核心服务器不同的是只读副本不参与关于集群拓扑的决策。丢失只读副本不会影响集群的可用性，除了图查询吞吐量的性能损失之外，也不会影响到集群的容错能力。

6.1.3　应用视图

从应用程序角度来看，集群的操作机制很有趣，但更重要的是考虑如何实际完成这些工作。在应用程序中，通常要从图数据库中读写数据。根据工作负载的性质，通常希望从图中读取时考虑到先前的写入，以确保因果一致性。



因果一致性是分布式系统中使用的一种一致性模型。它确保与因果相关的操作以相同的顺序被系统的每个节点看到。其结果是，集群中的客户端应用程序能做到读自己写（read-your-own-writes）的语义。读自己写语义确保客户端永远不会看到过时的数据，并享有集群容错和大规模的能力，还享有与单个数据库服务器一样的简单交互模式。

因果一致性确保数据能很容易地写入核心服务器（这可以保证数据的安全），并从只读副本中读到这些写入的数据。

在 Neo4j 驱动上构建的因果一致性如图 6-2 所示。



仅供非商业用途或交流学习使用

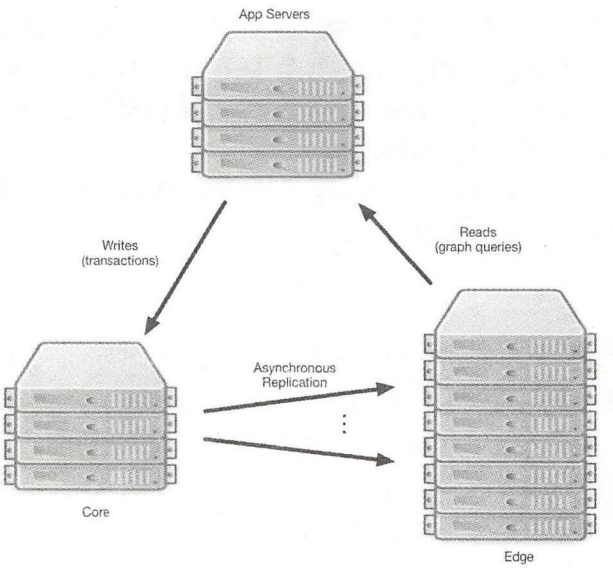


图 6-2　在 Neo4j 驱动上构建的因果一致性

在执行事务时，客户端可以请求书签（Bookmark），然后以该书签作为下一个事务的参数。使用书签功能，集群可以确保其中的服务器只有处理了客户端的书签事务后才能够运行其下一个事务。这提供了一个因果链，从客户的角度确保行为的正确性。

除了书签以外，剩余的工作都交由集群来处理。主要是由数据库驱动程序与集群拓扑管理器一起完成，以选择最合适的核心服务器和只读副本，从而提供高质量的服务。

在本小节中，我们从操作和应用程序开发两个高层面角度简要介绍了因果集群，并明白集群中的核心服务器负责数据的长效保存，而大量的只读副本负责分担图查询工作负载。究其原因，是 Neo4j 驱动程序极大地简化了这种功能强大的因果集群架构，抽象出集群拓扑结构以利于确保读层级的因果一致性。

6.1.4　因果集群的生命周期

本小节介绍因果集群的生命周期，包括：发现协议、加入集群、核心服务器和只读副本成员资格获取以及用于轮询、捕获和备份的协议到最后关闭集群。

在本小节中，我们将探讨一些关于集群如何操作的更深层次的内容。只有更深入理解集群是如何工作的，才能更好地设计、部署我们的生产系统，并解决系统中的故障。

探讨将遵循集群的整个生命周期。首先部署一个核心集群，并获取关键的架构基础作为集群的初步形式。然后添加只读副本，并学习如何将它们加入集群，如何发现并保持与核心服务器的联系。最后我们将看到在关闭只读副本和核心服务器之前，如何在在线集群环境中进行备份。

6.1.4.1　发现协议

发现协议是构建因果集群的第一步。它提供了现有核心集群服务器的一些提示，并使用这些提示初始化网络连接协议。因果集群发现协议如图 6-3 所示。



仅供非商业用途或交流学习使用

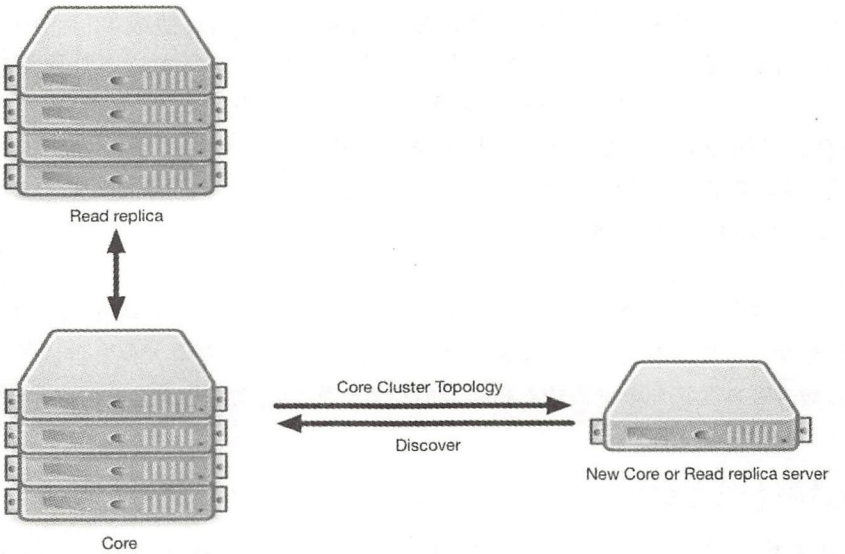


图 6-3　因果集群发现协议

从这些提示中，服务器将加入现有集群或形成自己的集群（不要担心形成分裂的集群，核心集群形成是安全的，因为它由 Raft 协议作支撑）。发现协议只支持从核心服务器到核心服务器和从只读副本到核心服务器的发现过程。



发现协议只针对核心服务器，而不管执行发现的是只读副本还是核心服务器。这是因为只读副本的数量众多而且并非稳定状态（只读副本数量可能会发生变化），而核心服务器的数量相对较少并相对稳定。

这些提示在 neo4j.conf 配置文件中表现为 initial_discovery_members 配置项，该配置项的值通常为点分十进制 IP 地址和通信端口。在执行发现的过程中，当前服务器将尝试与其他列出的服务器握手，在与另一个服务器握手成功时，当前服务器将发现整个集群的当前拓扑结构。

发现服务将继续运行在因果集群的整个生命周期，并用于维护可用服务器的当前状态，还能帮助客户端通过客户端驱动程序将查询转发到合适的服务器。

6.1.4.2　核心成员资格

如果是核心服务器正在执行发现操作，则一旦与现有核心服务器集群建立连接，它就会加入到 Raft 协议中来。



Raft 是一个分布式算法，是 Diego Ongaro 博士在 2014 年设计的，其目的是确保多个无共享服务器上的日志保持一致。

Raft 处理集群成员资格的方式是通过使其成为分布日志同步的一部分。加入集群需要将



仅供非商业用途或交流学习使用

集群成员资格条目插入到 Raft 日志中，然后复制到现有集群的其他成员日志中。一旦该条目成功地应用于 Raft 共识组（运行 Raft 算法的服务器）中的大部分成员，则更新它们的集群视图以吸收新加入的成员。新加入的核心服务器还必须在初始化其内部 Raft 实例时，从其他核心服务器获取自己的 Raft 日志。以上只是集群首次引导并且执行了很少操作的正常情况。如果核心服务器还需要获取其他服务器的图数据，则会出现一定时间的延迟，这对一个拥有大量图数据且长时间运行的集群而言，属于正常情况。



当一个实例想要建立向另一个实例的连接时，它取决于集群的当前状态和它本身是否有资格加入。要想有资格加入集群，该 Neo4j 实例必须存储与其他集群成员相同的数据库存储（可以允许此数据库的数据是老的或过时的），或者是没有数据库存储的新部署。

6.1.4.3 只读副本成员资格

当只读副本执行发现协议时，一旦它与任何可用的核心集群建立连接，它就会将自己添加到共享白板（Shared Whiteboard）中。只读副本添加到共享白板如图 6-4 所示。

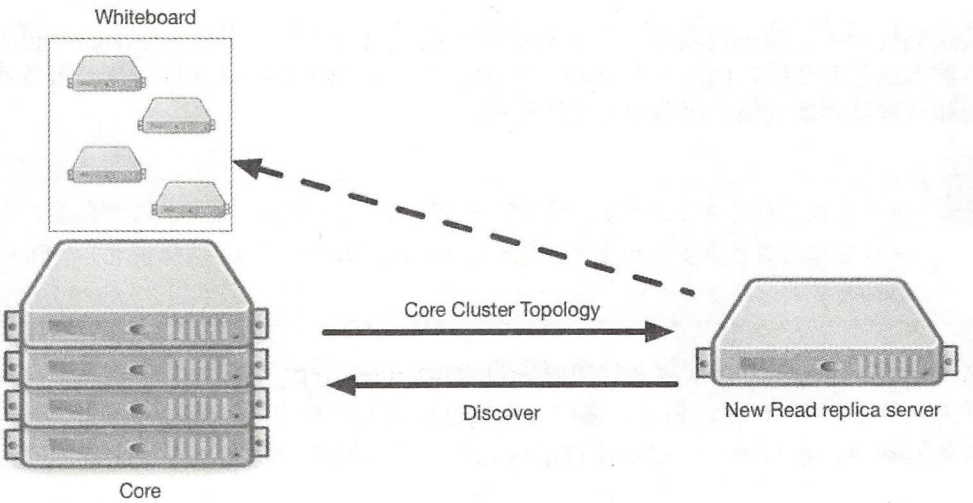


图 6-4 只读副本添加到共享白板

此白板实时地提供所有只读副本的视图，并用于最终用户应用程序的数据库驱动程序中的请求路由和集群的状态的监控。



只读副本不涉及 Raft 协议，也不影响集群拓扑。因此，在 Raft 协议之外的共享白板可以轻松地扩展只读副本的规模。

共享白板需要保持最新的状态，因为不断地有只读副本的加入或者离开集群。

6.1.4.4 通过 Raft 协议实现事务

一旦集群成功建立，每个核心服务器都会花时间处理数据库事务。更新操作会通过 Raft



仅供非商业用途或交流学习使用

协议在核心服务器之间可靠复制。更新操作以包含事务命令的 Raft 日志条目的形式呈现，再作用到图数据上。



Raft 协议的一个主要设计目标是易于理解，从而降低了协议实现中出现错误的可能性。同时也便于数据库操作人员理解因果集群中核心服务器这一概念。

因果集群中仅在核心服务器之间的通信采用了 Raft 协议，并对集群中的核心服务器划分为三类角色：Leader（领导者，只有一个）、Follower（追随者，通常多个）、Candidate（候选人，通常多个）。其基本思想来自“假象”民主社会的选举，Leader 是由民众投票选举产生。集群中核心服务器启动刚开始是没有领导者的，所有核心服务器的角色都为追随者，接下来，将进行选举，此时所有核心服务器都参与进来，且角色转换为候选人，一台核心服务器有且只能投一个为领导者，得票超过半数以上的将当选为领导者，并设定这届领导者的任期，从而选举结束，除领导者以外的其他候选人角色将转换为追随者并无条件地服从领导者领导。其中有一个概念为“任期”，在 Raft 协议中的术语为 Term。至此可以看出候选人角色是一个暂态，主要在选举期间出现，上述思想非常易于理解，因此，各种开源实现在很短时间内就蜂拥而现。三类角色的变迁图如图 6-5 所示。

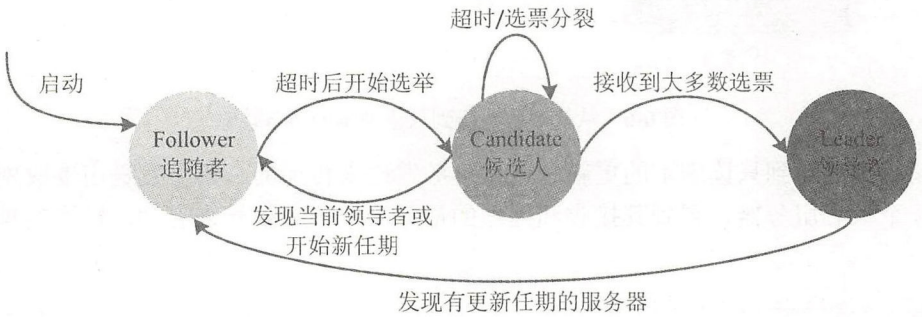


图 6-5　Raft 协议角色变迁图

Raft 协议的事务执行过程是，某时刻（逻辑时钟）Raft Leader 将事务附加到其本地日志的头部，并且要求其他实例执行相同的操作。当领导者（Leader）看到大多数的实例已经附加上此条目后，就可以将这个事务提交到 Raft 日志中。至此可以通知客户端应用程序事务已经安全地提交，因为在系统中已经有足够的冗余来容忍任何故障。

为了安全起见，在任何一个 Raft 协议实例中，只有一个领导者能够在给定的任期内来驱动协议的正常运转。领导者承担强制 Raft 日志排序和管理日志的责任。



Raft 协议描述了一个实例可以扮演的三个角色：领导者（Leader）、追随者（Follower）和候选者（Candidate）。这些角色是可以转变的，任何核心服务器都可以在集群的整个生命周期中扮演到上述三个角色。虽然从计算机学科的视角来看，理解这些状态是有意义的，但是实际操作人员无须过多地关注这些细节。

追随者根据领导者的日志变化来更新自身的日志。如果集群中的任何参与者怀疑领导者



仅供非商业用途或交流学习使用

出现故障，那么它可以通过进入候选者状态来发起领导者选举。在 Neo4j 核心服务器上这件事发生在毫秒级别时间内，默认为 500 毫秒。

无论哪个实例处于最佳状态（包括现有的领导者）都可以从选举中作为领导者出现。领导者的“最佳状态”先看最长运行时间，再看最长日志，最后看最高提交条目数。

在不丢失数据的情况下能按协议自动切换角色的能力，这个是在出现故障时事务操作能正常运转的前提。即使某个 Raft 实例出现故障，协议也能快速地从剩下的实例中找到一个最佳的替代者，从而保证数据最大限度地无丢失。这是非阻塞一致性协议的本质，Neo4j 因果集群为应用程序提供了高可用性。

6.1.4.5 跟踪协议（Catchup protocol）

只读副本一方面要处理图查询，另一方面要实现与核心服务器事务流更新的一致性。从核心服务器到只读副本的事务传输如图 6-6 所示。

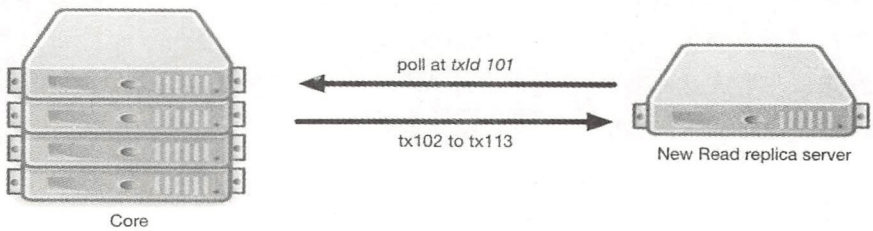


图 6-6 从核心服务器到只读副本的事务传输

从核心服务器到只读副本的更新是通过事务发送来传递的。事务发送由读取副本频繁轮询任意一个核心服务器，得到其接收和处理的最后一个事务编号来启动。轮询的频率是可设置的。



Neo4j 事务编号是严格递增的整数值。只读副本可以通过自身最后处理的事务编号与核心服务器的事务编号进行比较来确定事务是否已经在该只读副本上执行过。

如果只读副本的事务历史记录与核心服务器的事务历史记录之间存在较大差异，则上述轮询方式可能不会产生任何事务传递。这种情况是很普遍的，例如新的只读副本加入到一个已经运行了一段时间的集群或者是该只读副本已经停机了相当长一段时间，在这种情况下捕获协议意识到核心服务器和只读副本之间的内容相差太大，无法通过事务传输来弥补，于是执行回滚操作并直接将数据库存储内容从核心服务器复制到该只读副本。由于操作的是在线系统，因此在该复制操作完成的时候核心服务器中的数据可能已经更改，这时候只读副本可通过轮询请求来获取在复制数据期间遗漏的所有事务，以实现与核心服务器的数据一致。



由于直接从核心服务器复制数据的速度较慢，可能会使只读副本的数据状态远远落后于核心服务器，因为在这段时间里核心服务器已经发生了变化，这时可以通过事务日志跟踪协议来赶上核心服务器的进度。在这种情况下，只读副本服务器不断地与核心服务器进行重复捕获。



仅供非商业用途或交流学习使用

6.1.4.6　备份协议

在因果集群的生命周期中，操作人员希望备份集群数据以用于灾难恢复。备份是人为将在线系统与其最近的正常状态之间保持一定的差异，关键是这两个状态下数据都不存在故障。

有关如何备份 Neo4j 因果集群的详细操作方法，请参见“第 5 章 Neo4j 数据库管理”。

6.1.4.7　关闭只读副本

只读副本正常关闭时，将会调用发现协议从集群的共享白板视图中删除自身。发现协议还可以确保数据库的完全关闭和集群数据一致性，上述操作完成后才能关闭只读副本以便后续使用。

在非常规关机，例如断电时，维护集群视图的核心服务器会发现只读副本的连接已经突然断开。发现协议此时只隐藏只读副本的白板条目，如果只读副本在一定时间内没有重现，则其在共享白板中的内存会被回收。

非常规关机以后，只读副本可能不再具有完全一致的存储文件和事务日志。因此在后续重新启动时，只读副本将回滚这段时间的所有事务，以确保数据的一致性。

6.1.4.8　关闭核心服务器

正常关闭核心服务器需要通过 Raft 协议来处理。当一个核心服务器关闭时，协议会将成员资格条目附加到 Raft 日志上，然后其他的核心服务器重复同样的操作，一旦大多数的核心服务器都提交了成员资格条目，在逻辑上该核心服务器就可以离开集群并安全关闭。剩余的实例所组成的集群规模更小，因此容错能力会降低。如果离线的核心服务器在离开时扮演的正好是领导者角色，则在短暂的选举之后，另一个核心服务器成为领导者。

非正常的关机不会通知集群此核心服务器已经离开。因此集群中核心服务器的数量记录还是和以前相同（并不知道有一个核心服务器离线），而这个数字将会被应用于投票时的计算。例如，在一个 5 核心服务器的集群中有一个核心服务器非正常关闭，那么现在另一个核心服务器想要离线时需要 3/4 的成员同意，这比非常规关机之前 3/5 的要求更为严格。



当然，核心服务器发生故障时操作人员或监视脚本收到告警，以便在必要的时候人工干预集群。

6.1.5　创建一个新的因果集群

在本小节中，我们将学习如何部署一个全新的 Neo4j 因果集群。前两小节我们已经介绍了因果集群的概念和体系结构。在建立新的因果集群时，我们将学习如何设置每个 Neo4j 服务器的配置文件，然后学习如何部署一个具有 3 个核心实例的集群，3 个核心实例是形成一个安全的因果集群所需最少的服务器数量，并配三个只读副本，用来提供一定的横向可扩展性。从这个基本模式入手，就可以举一反三，配置出任何大小的因果集群。

仅供非商业用途或交流学习使用

6.1.5.1 下载并配置

从 Neo4j 官网下载 Neo4j 3.1.0 或以上的企业版，并且解压到 3 台目标机器上。然后分别配置每个机器上的配置文件。在这里我们主要编辑 neo4j.conf 文件，这是保存数据库配置的地方，此文件在数据库安装根目录下的 conf 目录中。

在三台独立的计算机上运行 Neo4j 时，实现一个因果集群的最基本配置是两个网络配置和三个集群配置。

位于“Network connector configuration”头下的两个配置分别是：

（1）dbms.connectors.default_listen_address

本机用于监听传入消息的地址或网络接口。如果取消此行的注释并且设置值为 0.0.0.0，则表明允许监听任何网络地址，即：dbms.connectors.default_listen_address=0.0.0.0。

（2）dbms.connectors.default_advertised_address

告知其他机器要连接的地址。在一般的情况下，设置为该服务器的公共 IP 地址。例如，如果 IP 地址为 10.108.192.211，则此项设置为：dbms.connectors.default_advertised_address=10.108.192.211。

在“Causal Clustering Configuration”头下的三个配置参数是：

（1）dbms.mode

数据库实例的运行模式。对于因果集群，有两种可能的运行模式：CORE 或 READ_REPLICA。我们将在三个实例上使用 CORE 模式，取消注释并设置为：dbms.mode=CORE。在另三个实例上使用 READ_REPLICA 模式，取消注释并设置为：dbms.mode=READ_REPLICA。

（2）causal_clustering.expected_core_cluster_size

启动时的初始集群大小。建立集群时需要在早期就实现稳定的核心服务器成员资格，以确保集群数据的安全读写操作。这个值是集群中核心服务器的数量。形成一个安全集群的最少核心服务器数量为 3，参见“6.1.2 操作视图”小节。此例中设置为：causal_clustering.expected_core_cluster_size=3。

（3）causal_clustering.initial_discovery_members

这个配置参数用于引导核心服务器或只读副本去初始发现核心集群成员。该值是以逗号为分隔的地址/端口对列表。在最简单的情况下，每个指定的网络地址将解析为正在运行的 Neo4j 核心服务器实例，而在极端情况下，这里列出的一个活动实例只能用于引导当前实例，当前实例通过发现协议发现其他可用的服务器。配置该值需要知道集群中核心服务器的地址，默认端口为 5000（还应该在此设置中包括本地计算机的地址）。本节示例配置为：causal_clustering.initial_discovery_members=server-1-IP:5000,server-2-IP:5000,server-3-IP:5000。



每个实例上的配置文件都需要这个配置参数，并且都相同。

仅供非商业用途或交流学习使用

6.1.5.2　启动 Neo4j 服务器

分别启动三台机器上的 Neo4j 服务器，启动顺序没有关系。示例操作如下：

```
server-1$  ./bin/neo4j start
server-2$  ./bin/neo4j start
server-3$  ./bin/neo4j start
```



如果您想了解服务器的启动过程，可以查看 logs/neo4j.logs 中的消息日志。当一个新的实例加入集群时，可能会暂时显示为不可用，因为在加入一个具有大量数据的集群时，新的实例可能需要几分钟的时间从集群复制数据。

现在可以正常地访问三个核心服务器并且检查它们的状态。从 Neo4j-shell 中发送查询命令：neo4j-sh (?)$　call dbms.cluster.overview()，可显示集群的状态和每个集群成员的信息。

6.1.5.3　添加核心服务器

添加一个新的核心服务器到集群的方法和“6.1.5.1 下载并配置”小节所述方法相同，只需要配置新添加的核心服务器的 neo4j.conf 配置文件，以及修改其他已存在集群中的核心服务器的 neo4j.conf 配置文件即可。

一旦重新配置完成，就可以重新启动服务器，新的核心服务器将自动加入到现有集群中。



如果要建立一个大规模的集群，最好使用脚本安装，因为手动安装是一个非常麻烦的事情并且容易出错。在 logs/neo4j.log 文件中可以查看安装调试的信息。

6.1.5.4　设置集群领导者角色偏好

因果集群中的核心服务器使用 Raft 协议来保证一致性和安全性。Raft 协议的实现细节是它使用领导者角色对基础日志加以排序，其他的实例为追随者，复制领导者的状态。在具体的实现中，这意味着对数据库的写操作是由当前为领导者角色的核心实例来排序的。

在某些情况下，用户可能希望主动防止某些实例被选为领导者。例如，在多数据中心的场景中，我们可能希望领导者是从某个特定地理位置中的核心服务器中选举产生，以确保领导者的效率和性能，如图 6-7 所示。在 Neo4j 因果集群中我们可以设置某个实例拒绝成为领导者（只能作为追随者），相关的配置参数是 neo4j.conf 文件中的 causal_clustering.refuse_to_be_leader，该配置项默认为 false，设置为 true 表示在领导者选举过程中不会成为领导者。设置一个数据中心的实例拒绝成为领导者如图 6-7 所示。

仅供非商业用途或交流学习使用

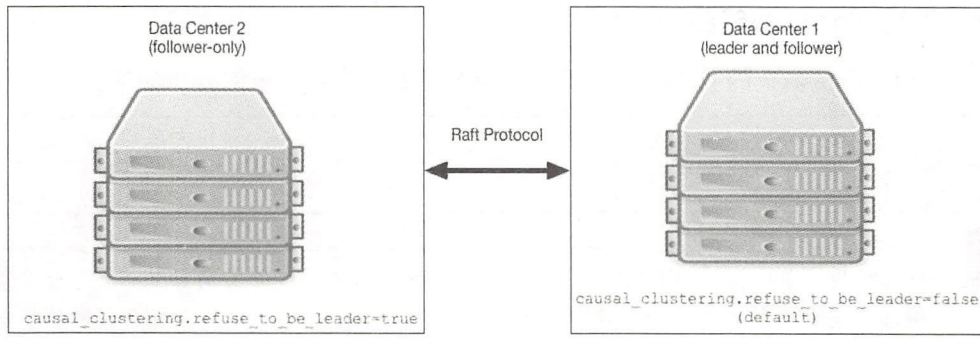


图 6-7　设置一个数据中心的实例拒绝成为领导者

为了最大限度地提高可用性，我们采用设置为拒绝成为领导者方式，而不是指定哪个实例作为领导者。最高的可用性是实现健康的实例在故障失效集群下还能正常运行，设置拒绝成为领导者方式，与指定领导者的方式相比具有更高的可用性，可以实现更安全的配置。



尽管表面上拒绝成为领导者的核心实例与只读副本类似。但是只读副本不参与事务处理，也不参与集群拓扑管理。而拒绝成为领导者的核心实例仍然按照 Raft 协议处理事务，以确保集群数据的一致性和安全性。

6.1.5.5　添加只读副本

添加只读副本与添加核心服务器类似。因为只读副本不参与集群的决策，因此配置参数更少。只读副本只需要知道核心服务器的地址，以便于绑定到这些地址上正常运行发现协议。一旦完成了发现过程，只读副本就会知道核心服务器的状态，并且可以选择一个合适的核心服务器进行数据复制。

在 neo4j.conf 文件的“Causal Clustering Configuration”部分中，需要更改一下设置。

（1）数据库的操作模式，须取消注释并且设置为：dbms.mode=READ_REPLICA。

（2）集群中核心实例的地址（与核心集群的相同），在本小节的示例中须设置为：causal_clustering.initial_discovery_members=server-1-IP:5000,server-2-IP:5000,server-3-IP:5000。

6.1.6　数据中心灾难恢复

本小节介绍如何从数据中心故障中恢复。具体来说，就是把少量幸存实例从只读（read-only）状态转换到可读/写（read-write）状态的实例组成的集群。

6.1.6.1　数据中心丢失

在多数据中心部署场景下，假如由于外部环境的作用使集群成员数量降低到总数的一半以下。最简单的情况就是一个 2+2 的部署，其中包含两个数据中心，每个数据中心包含两个实例。

在正常的情况下，这样的结构可以提供稳定的多数仲裁行为，其中四台机器中最快的三

台将执行用户的事务，如图 6-8 所示。

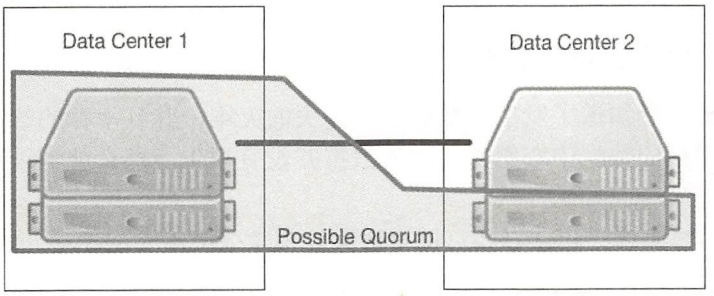


图 6-8　由四个核心实例组成的具有两个数据中心的部署

此时如果有一个数据中心由于灾难而变成脱机状态，在这种情况下不能达成多数仲裁。从而剩余的两个实例将变成只读状态。如图 6-9 所示。

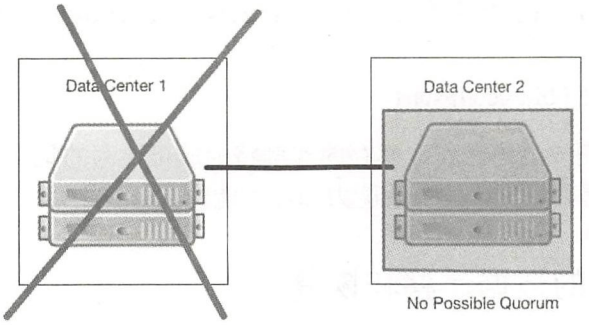


图 6-9　一个数据中心发生故障的情况



Neo4j 核心集群的事务处理基于 Raft 协议。Raft 需要集群中的大多数成员同意，才能执行事务以确保集群和数据的安全性。因此，当无法实现多数仲裁行为时，集群的剩余成员会自动转为只读状态。

数据中心可能在灾难中突然失效而非正常关闭，集群中其余核心服务器会仍然认为自己是原本大集群的一部分。这与实时数据中心的一个实例出故障不同，这种情况通常可以由集群底层中间件检测到，从而允许对集群进行自动重新配置。

但失效的是数据中心，集群是不会自动重新配置的。因为这种失效对其他成员来说是突然出现的，而剩余的每个实例只具有该集群的自己所处状态的部分视图，所以允许任何机器做出改变该集群的任何配置都是不安全的。

因此上述只剩下两个幸存核心服务器时，是不能形成仲裁所需法定人数的（3/4），集群也就不能正常工作。

但是假如知道了集群的实际情况，我们可以看到幸存的两台机器仍然可以形成一个非容错簇。

为了安全起见，在整个系统上具有完整可信视图的操作者和其他代理（例如触发脚本）

就需要尽快做出决定。在幸存的数据中心中，集群可以重新启动作为较小的配置群，同时保留提交到此时刻的所有数据。尽管切换期间用户可能无法使用数据库，但是不会丢失数据。

6.1.6.2 数据中心丢失后的恢复过程

假定我们已经完全知道了发生了数据中心丢失的灾难，并且幸存的数据中心已经转变成只读集群，此外幸存的数据中心集群成员的数据并没有损失，那么就可以从幸存只读集群来创建新的集群。

在确认上述情况之后，将幸存的只读集群恢复到完全可用（可读写）的过程如下：

（1）确保已经发生了数据中心丢失的灾难，并且可以访问正在运行的集群中幸存的成员，然后对集群中每个实例进行以下操作。

（2）运行 bin/neo4j stop 命令停止该数据库实例。

（3）更改 neo4j.conf 中的配置，使配置参数 causal_clustering.initial_discovery_members 包含现在可用实例的 IP 地址。

（4）更改 neo4j.conf 中的配置参数，设置为：causal_clustering.expected_core_cluster_size=2，即：有两个实例幸存。

（5）重新启动实例 bin/neo4j start。

一旦为每个实例完成上述操作，幸存的实例将形成可读写的集群。还可以再加入新的成员到这个集群中，以提高该集群的负载能力和容错能力。

6.1.7 因果集群的数据迁移和恢复

6.1.7.1 数据的迁移和恢复

除非 Neo4j 集群是全新的，否则需要从原有的单实例 Neo4j 数据库或者 Neo4j 集群迁移数据。详情参见“5.5 Neo4j 数据库的备份与恢复”一节。

6.1.7.2 从以前的版本升级

如果想从早期的 Neo4j 版本升级，我们可以在 neo4j.conf 配置文件中将 dbms.allow_format_migration 配置参数的值设置为 true。

6.1.7.3 离开集群

如果我们想把集群中的核心服务器降级为独立实例，可以在一个关闭的数据库上运行 neo4j-admin ubind [--database=<name>]命令来实现。ubind 命令只能使用在关闭的数据库实例上，在任何活动的数据库实例上执行该命令都会失败。

6.1.7.4 从数据中心的情况下恢复

有时可能想以离线的方式来迁移数据库，例如，从 Neo4j 社区版升级到企业版，或者将数据库从一个企业站点迁移到另一个企业站点。这些情况下我们可以使用 neo4j-admin 的 dump 和 load 命令。详情参见“5.6.3 Neo4j 数据转储和加载”小节。

6.1.8　因果集群的其他设置

1. causal_clustering.raft_advertised_address

该配置项为地址/端口对，是本机 Neo4j 实例向集群的其他成员通知它在核心集群内监听 Raft 消息的位置。例如，causal_clustering.raft_advertised_address =192.168.33.20:7000 将监听绑定在 192.168.33.20 地址上端口为 7000 的网络接口的集群通信。

2. causal_clustering.transaction_advertised_address

值为地址/端口对，指定本机 Neo4j 实例在哪里监听事务传输捕获协议中的事务请求。例如，causal_clustering.transaction_advertised_address =192.168.33.20:6001 将监听绑定在地址为 192.168.33.20 端口为 6001 的网络接口上的集群成员的事务。

3. causal_clustering.discovery_listen_address

发现协议所使用的地址/端口对。例如，causal_clustering.discovery_listen_address=0.0.0.0:5001 将监听 5001 端口处的任何集群成员的通信。

4. causal_clustering.raft_listen_address

值为地址/端口对，指定 Neo4j 实例将绑定到哪个网络地址和端口用以进行集群通信。此配置项必须与 causal_clustering.raft_advertised_addres 相配合，因为当前实例也会监听本机的 causal_clustering.raft_advertised_address 设置。例如，causal_clustering.raft_listen_addres=0.0.0.0:7000 将监听集群中任何网络接口在端口 7000 上的通信。

5. causal_clustering.transaction_listen_address

值为地址/端口对，指定 Neo4j 实例将绑定到哪个网络地址和端口来进行集群通信。此设置必须与 causal_clustering.transaction_advertised_addres 相配合，因为当前实例也会监听本机的 causal_clustering.raft_advertised_address 设置。例如，causal_clustering.transaction_listen_address=0.0.0.0:6001 将监听集群中任何网络接口在端口 6001 上的通信。

因果集群的运维方面可以参见“5.4 Neo4j 数据库运维与优化”一节。

6.2　高可用性集群

本节将对高可用性集群进行全面的介绍。首先介绍关于构成高可用性集群的不同组件，再详细介绍配置信息，并给出如何配置高可用性集群。

高可用性 Neo4j 集群采用了主从复制结构，从大的方面来说，集群可提供两个关键的能力：一个是在硬件故障时的应变能力和容错能力，另一个是扩展 Neo4j 读密集型数据场景的能力。

弹性和容错是指即使在网络或者硬件发生故障时，Neo4j 集群仍然可以继续提供服务的能力。如果集群中的一个节点坏了，或者网络连接断了，Neo4j 集群应该能够继续提供服务，而不是完全失去了服务的能力。

除了支持硬件容错之外，主从结构还有一个附加的好处，就是能提供在读密集型数据的场景下可横向扩展的能力。此处，读密集型数据场景指当有需求时具有高吞吐量的应用程序，可以在多个从实例中横向扩展读操作。还支持缓存分区，使得 Neo4j 高可用性集群比 Neo4j 单实例具有更大的负载处理能力。

6.2.1 高可用性集群概述

一个 Neo4j 高可用性集群由一个主实例和零个或多个从实例组成，集群中所有的实例都在其本地数据库文件中存有集群数据的完整副本。如图 6-10 所示，下面以三个实例的 Neo4j 高可用性集群来展开介绍。

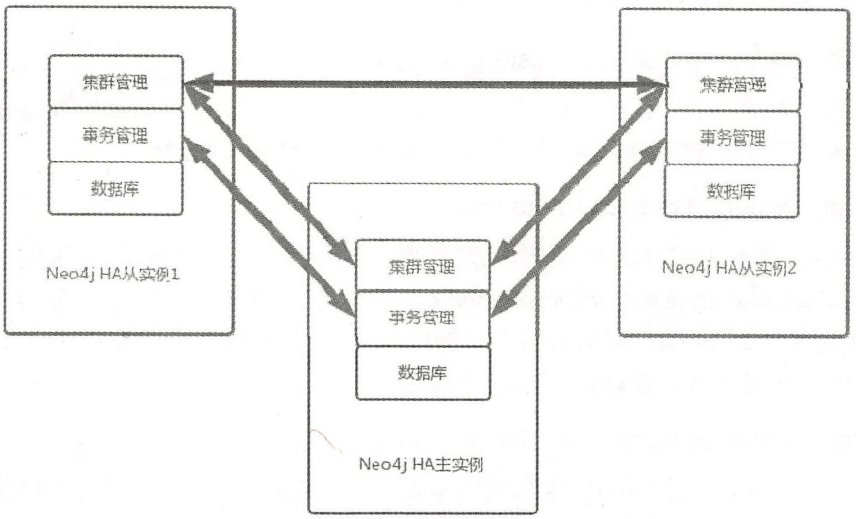


图 6-10 具有三个实例的 Neo4j 高可用性集群

图 6-10 中横线箭头代表的是协调每个数据库实例与其他成员之间的数据复制和选举管理。斜线箭头代表的是集群中从机与主机之间的通信，以使数据库保持最新。

6.2.1.1 仲裁实例

仲裁实例是一个在仲裁模式下运行的完整 Neo4j 实例，但是它只参与集群的通信，用于在选举出现僵局时进行仲裁，并不参与集群的其他操作（也不复制集群中的数据）。

6.2.1.2 事务传播

直接在主机上执行的写操作和在非集群模式下的方式一样。在主机上执行成功后，该事务将会被推送到集群中的从机上，并且这种方式是乐观的，意味着即使推送失败，也认为该事务的执行是成功的。

当在从机上执行写操作时，每个写操作首先与主机同步。此时主机和从机都会被锁定，当事务提交时，它首先在主机上提交，在主机上成功提交以后才会在从机上提交。为了确保一致性，在执行写操作时主机和从机的状态必须是最新的。从机的自动更新协议包括在主机和从机的通信协议中。

6.2.1.3　故障恢复

当集群中的一个实例变得不可用时（例如由硬件或网络故障引起），集群中的其他实例将会检测到此问题并将其标记为临时故障。在该实例发生故障以后如果可以重新实现和集群同步，那么该实例将会重新加入集群。

如果一个主机发生故障，那么在集群中会进行主机选举，如果一个从机被选中，那么它的角色将会从从机变成主机。当新的主机当选以后，它会向集群中的其他成员通报自己的身份。通常这个主机选举过程在几秒内完成，在这个时间内不能进行写操作。

1. 法定人数

集群必须具有法定人数才能通过选举产生新的主机。法定人数的定义为超过 50%的集群活动成员。设计一个高可用性集群的经验是：要容忍 n 个主机发生故障的集群需要 2n+1 个实例来满足选举条件的触发。因此最简单的高可用性集群需要三个实例，并且允许一个主机故障。

2. 选举规则

（1）如果主机发生故障或者在集群冷启动时，提交事务最多的节点将会被选为主机。

（2）如果主机发生故障，并且有两个从机的事务提交数量一样多，那么 ha.server_id 值较小的节点将会被选为主机。因为在集群中每个实例的 ha.server_id 是唯一的，这样就可以人为地设置哪个实例可以优先被选择为主机。

6.2.1.4　分支

数据分支可能会由以下两种方式引起：

（1）从机落后主机很多，然后离开或重新加入集群。这种类型的分支是无害的。

（2）重新发生了主机选举，旧的主机有一个或多个还未推送的事务，或从机在主机发生故障之前没有收到主机发出的事务，这样的分支是有害的，需要采取措施。

遇到这种情况最好的办法是数据库创建一个包含分支产生之前的数据库文件的目录，以便可以检查和解决出现的问题。

6.2.1.5　总结

- 写入事务可以在集群中的任何实例中执行。
- Neo4j 高可用性集群是具有容错功能的。
- 从机的写操作将先在主机上同步。
- 如果主机发生故障，则自动选出新的主机。
- 集群中会自动处理一个节点发生故障的情况，并且在它再次变得可用时还可以重新加入集群。
- 事务是具有原子性、一致性和持久性的，并且在最后才传递到从机。
- 如果主机关闭，那么任何正在运行的写事务将执行回滚，新事务将会阻塞或者执行

失败，直到主机变得可用为止。

- 读操作的负载能力会随着集群中的数据库实例增加而增加。

如果启动后不存在集群，意味着这是第一台试图连接集群的计算机，然后开始构建集群，并将这台计算机作为集群的主机。

所有的写请求都是通过主机实例完成的，尽管也有可能把请求发送到从机上执行。然而，这样要比直接由主机执行慢得多。从机在逻辑上为了确保同步会先写入主机，再写到自身。

在深入地讨论 Neo4j HA 特性之前，先来讨论具有三个实例的集群如何进行配置的。

6.2.2 建立 Neo4j 高可用性集群

HA 集群主要实现了高可用性和容错功能。搭建一个 Neo4j 高可用性集群主要可以按照以下步骤进行。

- 在每台机器上安装 Neo4j 企业版。
- 如果需要的话，可以配置作为仲裁实例运行的服务器。
- 在每个 Neo4j 实例上修改 Neo4j 的配置文件。
- 安装单个 Neo4j 实例。
- 按照以下两个小节的说明修改每个服务器上的配置文件。有些参数可以实现某种特定功能。不过最重要的三个配置参数分别是 dbms.mode、ha.server_id 和 ha.initial_hosts。

6.2.2.1 重要配置项

在启动 Neo4j 集群时，每个 Neo4j 实例按照配置项来关联其他实例。当一个实例需要建立到其他实例的连接时，能否加入集群取决于集群的当前状态和该实例是否有资格加入。为了保证具有加入资格，每个 Neo4j 实例必须拥有与集群的其他成员相同的数据库存储，或者是没有数据库存储的全新部署。

请注意，应为参与集群的实例显式配置 IP 地址或主机名。在没有指定的 IP 地址的情况下，Neo4j 将尝试找到一个有效的绑定接口，但是不推荐这样做。

dbms.mode 配置数据库的操作模式，需将它设置为：dbms.mode=HA。

ha.server_id 是每个实例的集群标识符。它必须是正整数，并且必须在集群中的所有 Neo4j 实例中是唯一的，例如，ha.server_id=1。

ha.host.coordination 是指 Neo4j 实例监听集群通信的端口，默认端口是 5001。例如，ha.host.coordination=192.168.33.22:5001 将 5001 设为集群通信端口。

ha.ininial_hosts 是用逗号隔开的地址/端口对列表，该列表指定一个 Neo4j 实例如何与其他的实例进行通信。当 Neo4j 实例启动时，将使用这些主机地址来查找和加入集群。当集群冷启动即没有集群可用时，数据库也是不可用的，直到 ha.initial_hosts 中列出的所有成员都在线并且彼此通信才可以恢复使用。一个好的做法是将集群中的每个实例的 ha.initial_hosts 都配置相同，以使集群中的各个成员可以快速连接上。例如 ha.initial_hosts = 192.168.33.21:5001, 192.168.33.22:5001, 192.168.33.23:5001 所有的监听端口都在端口 5001 上。

ha.host.data 是一个地址/端口对设置，用来指定 Neo4j 从机监听来自主机的事务，默认端口是 6001。ha.host.data 必须使用与 ha.host.coordination 不同的端口。例如，ha.host.data=192.168.33.22:6001 将监听来自端口 6001 的所有事务。

6.2.2.2　安装与配置

从 Neo4j 官方网站下载 Neo4j 企业版，解压缩到三台不同的机器上。按照如下配置过程配置集群。

分别设置三个实例名为 Neo4j-01、Neo4j-02、Neo4j-03。在 Neo4j-01 的服务器上配置 conf/neo4j.conf 文件。

```
# Unique server id for this Neo4j instance
# can not be negative id and must be unique
ha.server_id = 1
# List of other known instances in this cluster
ha.initial_hosts = neo4j-01.local:5001,neo4j-02.local:5001,neo4j-
03.local:5001
# Alternatively, use IP addresses:
#ha.initial_hosts = 192.168.0.20:5001,192.168.0.21:5001,192.168.0.22:5001
# HA - High Availability
# SINGLE - Single mode, default.
dbms.mode=HA
# HTTP Connector
dbms.connector.http.enabled=true
dbms.connector.http.listen_address=:7474
```

在 Neo4j-02 的服务器上配置如下文件：

```
# Unique server id for this Neo4j instance
# can not be negative id and must be unique
ha.server_id = 2
# List of other known instances in this cluster
ha.initial_hosts = neo4j-01.local:5001,neo4j-02.local:5001,neo4j-
03.local:5001
# Alternatively, use IP addresses:
#ha.initial_hosts = 192.168.0.20:5001,192.168.0.21:5001,192.168.0.22:5001
# HA - High Availability
# SINGLE - Single mode, default.
dbms.mode=HA
# HTTP Connector
dbms.connector.http.enabled=true
dbms.connector.http.listen_address=:7474
```

在 Neo4j-03 实例上修改配置文件：

```
# Unique server id for this Neo4j instance
# can not be negative id and must be unique
ha.server_id = 3
# List of other known instances in this cluster
ha.initial_hosts = neo4j-01.local:5001,neo4j-02.local:5001,neo4j-
```

```
03.local:5001
# Alternatively, use IP addresses:
#ha.initial_hosts = 192.168.0.20:5001,192.168.0.21:5001,192.168.0.22:5001
# HA - High Availability
# SINGLE - Single mode, default.
dbms.mode=HA
# HTTP Connector
dbms.connector.http.enabled=true
dbms.connector.http.listen_address=:7474
```

配置完成以后重新启动三台服务器上 Neo4j 数据库。

6.2.2.3 设置仲裁实例

Neo4j 高可用性集群的典型部署是使用三台机器来提供容错和读可扩展性。虽然为了支持故障迁移功能，Neo4j 高可用性集群至少需要 3 个实例（以防主机变得不可用），但并非所有实例都需要托管集群的数据，我们还可以部署一种称为仲裁实例的 Neo4j 服务器，它也被视为集群的一员。它的作用是参加主机选举，其目的是打破选举过程中的僵局。这样我们也可以部署这样的集群，集群中有两个 Neo4j 数据库实例和一个仲裁实例，并且仍然能够容忍三个实例中任何一个的故障。

仲裁实例的 neo4j.conf 的配置和其他实例的配置基本相同。只是将 dbms.mode 选项的值变为 ARBITER。启动 Neo4j 仲裁实例的方法和启动一个普通实例也一样。停止仲裁实例的服务也和其他 Neo4j 实例完全相同。

```
neo4j_home$ ./bin/neo4j start
```

6.2.3 状态信息端点

Neo4j 高可用性集群常见的操作是将所有写入请求定向到主机中，从机只进行读取操作，在集群中对读操作做到负载均衡，同时支持故障迁移。实现这个功能最常见的方法是在集群上加入一个负载均衡器，可以选用 HA Proxy（一个支持高可用性、负载均衡 TCP 和 HTTP 开源应用程序代理软件）[21]。这里的负载均衡器使用 HTTP 端点来发现哪个是主节点并且直接承担写操作负载。在本小节中，我们将解释这些 HTTP 端点的作用。

6.2.3.1 状态信息端点介绍

每个高可用性集群的实例都有关于其在集群中状态的 3 个端点。每个端点都是可用的，具体的状态取决于你的负载均衡的需求和生产设置。3 个端点分别是：

- /db/manage/server/ha/master
- /db/manage/server/ha/slave
- /db/manage/server/ha/available

/master 和/slave 可以将写入和非写入请求引导向特定的实例。/available 端点一般情况下

[21] http://www.haproxy.org/

会将任意请求指向可用于事务处理的实例。

要查询端点状态信息，请对任一端点执行 HTTP GET 操作，将返回如表 6-1 所示的对应内容。

表 6-1　状态信息端点详情

端点	实例状态	返回状态码	正文文本
/db/manage/server/ha/master	Master	200 OK	true
	Slave	404 Not Found	flase
	Unknown	404 Not Found	Unknown
/db/mansge/server/ha/slave	Master	404 Not Found	flase
	Slave	200 OK	true
	Unknown	404 Not Found	Unknown
/db/manage/server/ha/available	Master	200 OK	master
	Slave	200 OK	slave
	Unknown	404 Not Found	Unknown

6.2.3.2　运行

从命令行执行请求的最常用方法是使用 Curl。在没有参数的情况下，Curl 将对提供的 URL 执行 HTTP GET 操作，并且返回正文文本信息。如果还想获得响应代码内容，只需添加-v 标志来获取详细输出。

在正在运行的主机上发送端点请求命令及其详细输出：

```
#> curl  -v http://localhost:7474/db/manage/server/ha/master
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 7474 (#0)
* Server auth using Basic with user 'neo4j'
> GET /db/manage/server/ha/master HTTP/1.1
> Host: localhost:7474
> Authorization: Basic bmVvNGo6NjU0MzIx
> User-Agent: curl/7.47.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Fri, 24 Feb 2017 07:17:50 GMT
< Content-Type: text/plain
< Access-Control-Allow-Origin: *
< Transfer-Encoding: chunked
< Server: Jetty(9.2.9.v20150224)
<
* Connection #0 to host localhost left intact
true
```

在正在运行的从机上发送端点请求命令并且不输出详细信息：

```
#> curl localhost:7474/db/manage/server/ha/slave
false
```



UNKNOWN 状态表示当前 Neo4j 实例既不是主机也不是从机。例如，可能是该实例正在进行状态转换（在故障状态下从从机转换成主机），或者该实例是仲裁实例。如果返回 UNKNOWN 状态，客户端不能将该实例视为主机或者从机，而应该选择集群中的另一个实例，并等待该实例转换至非 UNKNOWN 状态，或者通过系统管理进行恢复。

6.2.4 HAProxy 用于负载均衡

在 Neo4j 高可用性架构中，集群通常由负载均衡器管理。在本节中我们将探讨如何设置 HAProxy 以在高可用性集群上执行负载均衡。

首先在 Linux 环境下安装 HAProxy。

6.2.4.1 为 Bolt 协议配置 HAProxy

在 Neo4j 高可用性集群的部署中，HAProxy 将配置两个端口，一个用于将写入操作路由到主机，另一个用于从机读操作的负载均衡。每个应用程序都有两个驱动程序实例，一个连接到主机端口用于执行写操作，一个连接到从机端口用于执行读操作。

首先我们进行通信模式和超时时间的设置。使用以下设置表示：如果一个服务器或者客户端空闲超过两小时，那么该客户端或者服务器会终止连接。

```
defaults
    mode          tcp
    timeout connect   30s
    timeout client     2h
    timeout server     2h
```

设置执行写操作的驱动程序将要连接到：

```
frontend neo4j-write
    bind *:7680
    default_backend current-master
```

设置指向当前主机的后端配置：

```
backend current-master
    option  httpchk  HEAD  /db/manage/server/ha/master HTTP/1.0
    server db01 10.0.1.10:7687 check port 7474
    server db02 10.0.1.11:7687 check port 7474
    server db03 10.0.1.12:7687 check port 7474
```

设置执行读取操作的驱动程序将要连接到的位置：

```
frontend neo4j-read
    bind *:7681
    default_backend slaves
```

设置一个以循环方式指向从机的后端配置：

```
backend slaves
    balance roundrobin
    option  httpchk  HEAD  /db/manage/server/ha/slave HTTP/1.0
    server db01 10.0.1.10:7687 check port 7474
    server db02 10.0.1.11:7687 check port 7474
    server db03 10.0.1.12:7687 check port 7474
```

注意：从机的后端配置与主机的后端配置相同。

然后将上述的所有配置放到一个文件中，我们就可以得到一个基本可运行的 HAProxy 配置文件，它采用 Bolt 协议为应用程序提供负载均衡功能。

6.2.4.2　为 HTTP API 配置 HAProxy

HAProxy 的配置方式有很多种，完整文档可以参阅 HAProxy 的官方网站。

在这里我们只演示如何配置 HAProxy 以将请求负载均衡到三台 Neo4j 服务器上。我们只需要修改以下配置文件/etc/haproxy/haproxy.cfg。

```
global
    daemon
maxconn 256
defaults
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms
frontend http-in
    bind *:80
    default_backend neo4j
backend neo4j
    option httpchk GET /db/manage/server/ha/available
    server s1 10.0.1.10:7474 maxconn 32
    server s2 10.0.1.11:7474 maxconn 32
    server s3 10.0.1.12:7474 maxconn 32
listen admin
    bind *:8080
    stats enable
```

HAProxy 现在可以通过如下方式启动：

```
/usr/sbin/haproxy  -f  /etc/haproxy/haproxy.cfg
```

现在你可以连接到 http:// <ha-proxy-ip>:8080/haproxy?stats 来查看状态信息。默认用户名密码为 admin。

6.2.4.3　优化读写

Neo4j 支持 HAProxy（或者其他的负载均衡器）使用 HTTP 响应代码区分集群实例是否可用的功能。在上面的例子中，我们使用/available 端点将请求定向到可以执行事务处理的机器，也可以将请求仅定向到从机/slave，或者仅定向到主机/master。如果在应用程序中可以分清写请求和只读请求，那就可以使用两个独立的逻辑负载均衡器：一个用来将所有写请求定

向到主机，一个将所有读请求定向到从机。在 HAProxy 中可以通过添加多个后端服务来构建逻辑负载均衡器。

虽然 Neo4j 实例也允许从机代理写操作，但这种代理不但会占用从机的资源，还会增加写操作的延迟。同样的主机也不应该代理大量的读操作（因为在 HA 集群中写操作主要是在主机上进行的），因此我们应该尽量地保证主机只进行写操作，从机只进行读操作，以发挥 HA 集群的最大性能。

下面是将请求只定向到从机的配置：

```
global
    daemon
    maxconn 256
defaults
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms
frontend http-in
    bind *:80
    default_backend neo4j-slaves
backend neo4j-slaves
    option httpchk GET /db/manage/server/ha/slave
    server s1 10.0.1.10:7474 maxconn 32 check
    server s2 10.0.1.11:7474 maxconn 32 check
    server s3 10.0.1.12:7474 maxconn 32 check
listen admin
    bind *:8080
    stats enable
```



实际中，在从机上执行写操作是不常见的。虽然将数据写入从机有利于数据在两个位置持久化（主机和从机），但这是有代价的，从机必须立即变得和主机一致，此时从机需要丢弃正在执行的事务，然后同步与主机之间的差异。这将比在主机上执行写操作再推送到从机要麻烦得多。

6.3 集群的备份与恢复

将 Neo4j 数据库备份到远程或离线存储是企业级部署的一部分。Neo4j 支持完全备份和增量备份。对于单实例数据库，高可用性集群和因果集群，备份过程是相同的。详情请见“5.5 Neo4j 数据库的备份与恢复”一节的相关内容。

6.4 Neo4j 单实例和高可用性集群版本升级

本节介绍 Neo4j 单实例和高可用性集群的升级。为了方便体验新版本带来的新特性，并保证已经在使用的应用程序不至于遭到破坏。Neo4j 还提供了从低版本向高版本升级的功能。

6.4.1 单实例升级

6.4.1.1 从 Neo4j 2.x 版本升级

（1）如果数据库正在运行，请关闭数据库。

（2）创建数据库目录的备份副本。如果使用的是在线备份工具，请确保备份已经成功完成。

（3）安装 Neo4j 3.1.0。

（4）检查先前版本的配置文件中的设置，并将任何自定义设置迁移到 3.1.0 版本。由于 Neo4j 2.x 与 Neo4j 3.x 之间的配置文件差别较大，因此可以使用 2.x-config-migrator 迁移配置文件，找到 2.x-config-migrator 的工作目录，使用命令 java -jar 2.x-config-migrator.jar path/to/neo4j2.3 path/to/neo4j3.1.0 可以完成配置文件的迁移。最后查看所有的警告日志，手动编辑不正确的设置。

（5）从旧版本的数据库导入数据，输入命令为：neo4j-admin import --mode=database –database =<database-name> --from=<source-directory>。

（6）如果默认数据库不为 graph.db，在 neo4j.conf 配置文件中将配置参数 dbms.active_database 设为所需要的数据库名。

（7）将 Neo4j 3.1.0 的 neo4j.conf 配置项 dbms.allow_format_migration 的值设置为 true。如果不修改此配置项，则无法启动数据库服务。

（8）启动 Neo4j 3.1.0。

（9）数据库升级操作将在启动期间进行。

（10）有关升级和进度指示信息记录在 debug.log 文件中。

（11）当升级完成后，dbms.allow_format_migration 应设置 false 或删除。

（12）最好在升级后立即进行完全备份。

6.4.1.2 从 Neo4j 3.x 版本升级

（1）如果数据库正在运行，请关闭数据库。

（2）创建数据库目录的备份副本。如果使用的是在线备份工具，请确保备份已完成。

（3）安装 Neo4j 3.1.0。

（4）检查先前版本的配置文件中的设置，并将任何自定义设置导入 3.1.0 版本。

（5）如果使用默认的数据目录，直接将其从旧的版本复制到新的版本即可。如果数据库存储在自定义位置，配置 dbms.directories.data 指向这个自定义位置。

（6）如果默认数据库不为 graph.db，将 neo4j.conf 配置参数 dbms.active_database 改为实际的数据库名。

（7）将 Neo4j 3.1.0 的 neo4j.conf 配置项 dbms.allow_format_migration 的值设置为 true。如果不修改此配置项，则无法启动数据库服务。

（8）启动 Neo4j 3.1.0。

（9）数据库升级操作将在启动期间进行。

（10）有关升级和进度指示信息记录在 debug.log 文件中。

（11）当升级完成后，dbms.allow_format_migration 应设置 false 或删除。

（12）最好在升级后立即进行完全备份。

6.4.2 从高可用性集群升级

将 Neo4j 高可用性集群升级到 Neo4j 3.1.0 需要遵循特定的过程，以确保集群的一致性，使所有的实例在升级后都能够加入集群。

1. 备份 Neo4j 数据库

详情请见“5.5 Neo4j 数据库的备份与恢复”一节的相关内容。

2. 关闭集群

逐个关闭从机，最后关闭主机。

3. 升级主实例

（1）在主服务器上安装 Neo4j 3.1.0，保持数据库目录不变。

（2）将 neo4j.conf 中的 dbms.mode 配置项设置为 SINGLE，即：禁用 HA 模式。

（3）按照 Neo4j 单实例的升级方法升级。

（4）升级完成后再次关闭 Neo4j。

（5）重新将 neo4j.conf 中的 dbms.mode 配置项设置为 HA，即：启用 HA 模式。

（6）对 Neo4j 数据库进行完全备份。

4. 升级从实例

（1）删除数据库目录。

（2）安装 Neo4j 3.1.0。

（3）检查先前版本的配置文件中的设置，并将任何自定义设置导入 3.1.0 版本。

（4）如果默认数据库不为 graph.db，将 neo4j.conf 配置参数 dbms.active_database 改为实际的数据库名。

（5）从主实例复制安全配置，因为这些配置不会自动传播。

5. 重新启动集群

启动主实例，然后逐一启动从实例，一旦从实例加入集群，它将从主实例同步数据。

仅供非商业用途或交流学习使用

第 7 章

◂ Neo4j 应用案例 ▸

自 Neo4j 诞生以来，它已经在全球众多企业和机构获得了较为广泛且成功的应用，深受客户喜爱。本章主要对 Neo4j 目前的客户应用实践进行梳理和汇总，并着重介绍 Neo4j 的 5 个较为典型的应用案例。这样，可以让读者对 Neo4j 的具体应用价值有一个直观的感受和认识，以便启发读者将 Neo4j 应用到工作和学习的领域中去。本章共分 6 节，分别是：

- Neo4j 应用案例概述
- 欺诈检测
- 科研导图
- 电子邮件监测
- 工商企业图谱
- 社交网络

7.1 应用案例概述

从系统科学的视角来看，世界是由各种系统构成的，而系统又由系统的各个组成部分及其之间的联系组成。从这个层面，便能直接地将系统及其之间的联系映射为数学图论中的节点与关系上来，从而运用图论对世界进行直观建模，图数据库技术以图论为根基，也可以说是表达多姿多彩世界的基础性、通用性“语言”。这种“语言”描述出来的仿真系统与原系统相比具有“高保真”的特性，与人们通常对系统的认知是一致的，并且非常直观、自然、直接和高效，不需要中间过程的转换和处理——这种中间过程的转换和处理往往把问题复杂化或者漏掉很多有价值的信息。正是由于图数据库技术可以直接描述各种复杂的现实世界系统，才使其具有广泛的适用性和更高的应用价值。

事实上，Neo4j 已经成功“俘获”了大量客户，并且客户数量和应用领域还在不断增长之中。这些客户包括思科、惠普、沃尔玛、领英、阿迪达斯和 FT 金融时报等国际知名企业或机构，具体可以参见图 7-1。Neo4j 客户的行业分类目前主要集中在社交网络、人力资源与招聘、金融、保险、零售、广告、电子商务、物流、交通、IT、电信、制造业、打印、文化传媒和医疗等领域（如表 7-1 所示）。上述大量的 Neo4j 客户在未采用 Neo4j 产品之前反复抱怨原有产品的不足，有待新产品去解决和实现，通称为：痛点，表 7-2 归集汇总了部分 Neo4j 客户的主要痛点。

仅供非商业用途或交流学习使用

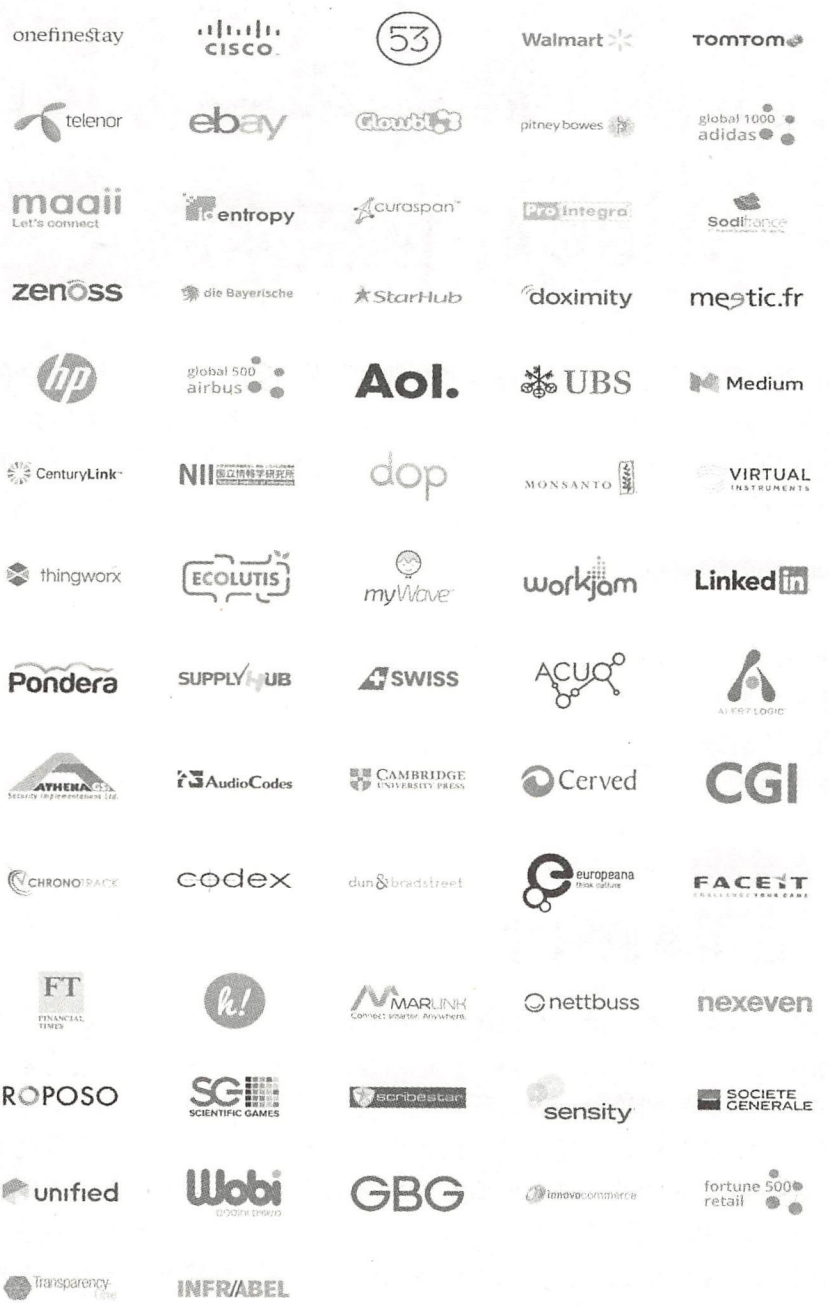


图 7-1　Neo4j 的客户

Neo4j 成功的典型案例主要包括：优化客户的个性化体验、实时推荐、主数据管理（Master Data Management ，MDM）等方面。例如，沃尔玛现在使用 Neo4j 来了解在线购物者的行为，快速查询客户过去的购买信息，及时捕获客户当前在线访问中显示的任何新的兴趣点，这对于提供实时推荐是至关重要的，关系数据库的查询会非常复杂，而图数据库则能轻松胜任。又比如 Pitney-Bowes 在提升客户服务中遇到了主数据管理问题：（1）有关客户

仅供非商业用途或交流学习使用

的数据分散在内部会计与计费系统、物流与交付系统、客户关系管理与销售管理系统等，而这些系统是公司发展过程中独立创建的，由此以数据孤岛的形式展现；（2）在不同业务环节中，利用孤立的局部数据对客户的不同定义导致了低质量的客户画像，几乎没有可预测性；（3）每个不同的信息孤岛已经针对不同任务进行了局部优化，但这些优化并不能突破问题的瓶颈，即无法提供全局优化的能力。Pitney-Bowes 凭借其领先的主数据管理系统，通过围绕 Neo4j 图数据库构建其 MDM 系统，避免了缓慢和不够灵活的关系数据库管理系统（RDBMS，Relational Database Management System）的缺陷，或不够丰富的非关系型数据库（NoSQL）解决方案，从而超越了其竞争对手。

本节通过客户代表、客户所在行业、客户痛点分析、Neo4j 优势等案例信息对 Neo4j 的 20 多个应用案例进行梳理和总结。希望读者能从这些案例中获得启发，将 Neo4j 应用到自己所从事的领域并做出有价值的创新性工作。

表 7-1　Neo4j 客户行业分布情况

Neo4j 应用行业分类	Neo4j 典型客户
社交网络、人力资源与招聘	Glowbl、megree、Gamesys、InfoJobs
金融、保险	Cerved、Wobi、Die Bayerische
零售、广告、营销、电子商务	Adidas、Qualia、Walmart、eBay
国际物流、消费品、物流	Global 500 Logistics、TRANSPARENCY-ONE
交通（旅行）	Wanderu
IT、软件、技术、电信	WineDataSystem、migRaven、Pitney Bowes、LinkedIn China、Telenor（挪威电信）
制造业	SchleichGmbH
打印、媒体、网页服务、音乐	BILLES、ICIJ、Wazoku、Musimap
医学、生命科学	Candiolo Cancer Institute (IRCC)

表 7-2　Neo4j 客户痛点分析

序号	企业名称	痛点分析或挑战
1	migRaven	授权和访问控制
2	adidas	提供个性化体验所需的数据分布在各种信息孤岛上
3	BILLES	增加在线客户；必须能够处理大量的小打印订单；大量的收购导致了 IT 系统的拼凑
4	Cerved	提高计算效率和快速识别，直接或间接控制公司的人员；获取大数据网络分析的顶尖技术
5	Die Bayerische	过时的管理系统和不同的数据格式；创建标准化数据框架
6	ICIJ	帮助记者打破复杂的瑞士漏洞数据，以获得更好的调查性新闻
7	Candiolo Cancer Institute (IRCC)	关系数据库没有为所需的多个功能提供足够的灵活性
8	LinkedIn China	尽可能快地启动社交网络平台，同时为重要的用户和功能增长留出空间，并搜索大量数据，而无性能问题
9	Musimap	要映射所有音乐标题，每个具有 55 个加权描述标准，以允许深入处理和实时推荐
10	Qualia	原始产品仅被优化以跟踪一个设备上的用户行为
11	SchleichGmbH	在产品数据网络中需要更大的可扩展性和灵活性
12	TRANSPARENCY-ONE	管理和搜索大量数据，没有性能问题
13	Wanderu	帮助消费者在美国的旅行找到和预订城市间公共汽车和火车

仅供非商业用途或交流学习使用

（续表）

序号	企业名称	痛点分析或挑战
14	WineDataSystem	没有现有的参考资源，大量的信息和问题；关于访问的方便性和用户的灵活性
15	Wobi	快速分析大量的“整个客户”信息
16	eBay	支持大规模的复杂路由查询，具有快速和一致的性能
17	Global 500 Logistics	时时刻刻都在产生地理位置路由信息，业务需要这些具有复杂关联关系的位置信息来支持，这导致传统关系型数据库面临严重挑战
18	Glowbl	将所有可能的社交网络汇集在一起，以图的形式表示所有联系人，并实时管理这些联系人及其互动关系
19	InfoJobs	建立新的门户，模拟求职者的潜在职业道路
20	megree	提供这些连接的关系和强度的整体视图
21	Pitney Bowes	通过构建下一代工具，获得 360 度的客户洞察力，获得竞争优势
22	Walmart	为客户提供最佳的网络购物体验
23	Telenor	在 Telenor 的在线自助服务管理门户的背后，您可以找到负责管理客户组织结构的协议，订阅和访问他们的商业移动订阅的中间件

7.2 欺诈检测

本节重点介绍 Neo4j 在银行、保险、电商等领域反欺诈方面的应用[22,23]。

银行和保险公司每年都会因为各种各样的欺诈行为导致数十亿美元的巨额损失。尽管传统的欺诈检测方法在减少欺诈损失方面起到了非常重要的作用，但我们发现越来越复杂的欺诈者已经开发出了多种新的合谋方法和构建假身份来规避传统反欺诈技术的识别。

图数据库为精准揭露“欺诈环”和其他复杂欺诈提供了新的方法和工具，并且能够实时阻止高级欺诈情况发生。尽管没有完美的欺诈预防措施，但是通过超越单个数据点并让多个节点进行连接，仍有改进反欺诈系统的机会，通常这些连接被忽视但又是最有价值的反欺诈线索。

理解节点数据之间的连接以及从这些连接中解读出新的价值有时候并不需要收集更多的数据，只需要用“图”重新定义问题并以新的方式考察问题，即可从现有数据中获得新的见解。图的最大特点是表达节点之间的相关性，这与其他大多数查看数据的方式是不同的。图数据库可以发现使用传统表示（如表）难以检测的“隐形”模式，从这个角度来讲，它就是在“反隐形”技术上的颠覆性突破。现在越来越多的公司正尝试使用图数据库来解决数据之间的各种连接问题，其中就包括欺诈检测这一大类应用。

本节主要讨论三种常见且极具破坏力的欺诈类型：第一方银行欺诈、保险欺诈和电子商务欺诈。尽管这三种欺诈类型不同，但它们都具有一个非常重要的共同点，那就是欺诈依赖

[22] https://Neo4j.com/resources/fraud-detection-white-paper/?ref=solutions
[23] https://Neo4j.com/graphgist/9d627127-003b-411a-b3ce-f8d3970c2afa?ref=solutions

仅供非商业用途或交流学习使用

于间接层且它们可以通过连接分析被揭示出来。在每个类型案例中，图数据库显著地改进了现有欺诈检测方法，使高级欺诈难以“隐身”。

7.2.1　第一方银行欺诈

第一方欺诈（First-Party Fraud）又叫真名欺诈（True Name Fraud）或身份欺诈（Identity Fraud），其本质是使用本人或他人真实身份或编造、伪造身份进行欺诈，其花样不断翻新，比如：“无意愿支付（No Intent to Pay）”“合成身份（Synthetic Identity）”和“亲友身份欺诈（Identity Fraud by Friends and Family）”等。具体到银行领域就称为：第一方银行欺诈（First-Party Bank Fraud），包括信用卡欺诈、贷款欺诈、透支欺诈、无担保银行信用额度欺诈，涉及欺诈者申请信用卡、贷款、透支和无担保的银行信用额度。这些欺诈的本质是通过信用卡、贷款等工具从银行骗取资金，但根本没有偿还的意愿。这是全球各银行机构面临的一个严重的难题。

美国银行业每年在第一方银行欺诈上的损失高达数百亿美元，据估算该损失占美国消费者信贷总额的四分之一甚至更高[24]。据进一步估计，在美国和欧洲有 10%~20%的无担保坏账被错误分类，实际上都属于第一方银行欺诈[25]。导致这些惊人的巨额损失的因素主要有两个：一是欺诈者的行为非常接近合法客户的行为，在他们没有跑路之前很难被发现；二是他们往往是有组织犯罪，并通过操纵者和参与者之间的关系形成难以被传统反欺诈技术发现的欺诈环。由此，当这些特征造成潜在的危害时，也同样表现出能够采用基于图的方法检测出这些欺诈行为。

每个第一方银行欺诈串通背后的具体细节因操作手法而不同，下面是欺诈环通常的操作模式：

- 一组两个或两个以上的人组织成一个欺诈环。
- 欺诈环共享合法联系人的一部分信息，例如电话号码和地址，通过它们的不同组合创建多个合成身份。
- 欺诈环中的人员使用这些合成身份打开账户。
- 新的账户被添加到原来的账户：无担保的信用额度、信用卡、透支保护、个人贷款等。
- 账户使用正常，定期购买和及时付款。
- 由于观察到负责任的信用行为，银行随着时间的推移增加了循环信用额度。
- 突然有一天环“消失”了，并且他们的行为步调一致，所有人都最大化使用了信用额度后跑路了。
- 有时欺诈者会进一步使用假的检查，在前一步骤之前使所有的余额归零，这必然会导致危害加倍。
- 从收集这些欺诈的过程来看，却根本找不到这些欺诈者。
- 于是，这些无法收回的债务只能被注销。

[24] http://www.experian.com/assets/decision-analytics/whitepapers/first-partyfraud-wp.pdf
[25] http://www.businessinsider.com/how-to-usesocial-networks-in-the-fight-against-first-party-fraud-2011-3

仅供非商业用途或交流学习使用

为了说明这种情况，让我们采取一个小的 2 人圈环，以创建合成身份：

Tony Bee 住在 123 NW 1st Street，San Francisco，CA 94101（他的真实地址），并获得一个预付费电话 415-123-4567；Paul Favre 住在 987 SW 1st Ave，San Francisco，CA 94102（他的真实地址），并得到一个预付费电话 415-987-6543。

仅利用共享的电话号码和地址这 2 条数据，即可组合创建出 2×2 = 4 个具有假名的合成身份，如图 7-2 所示。

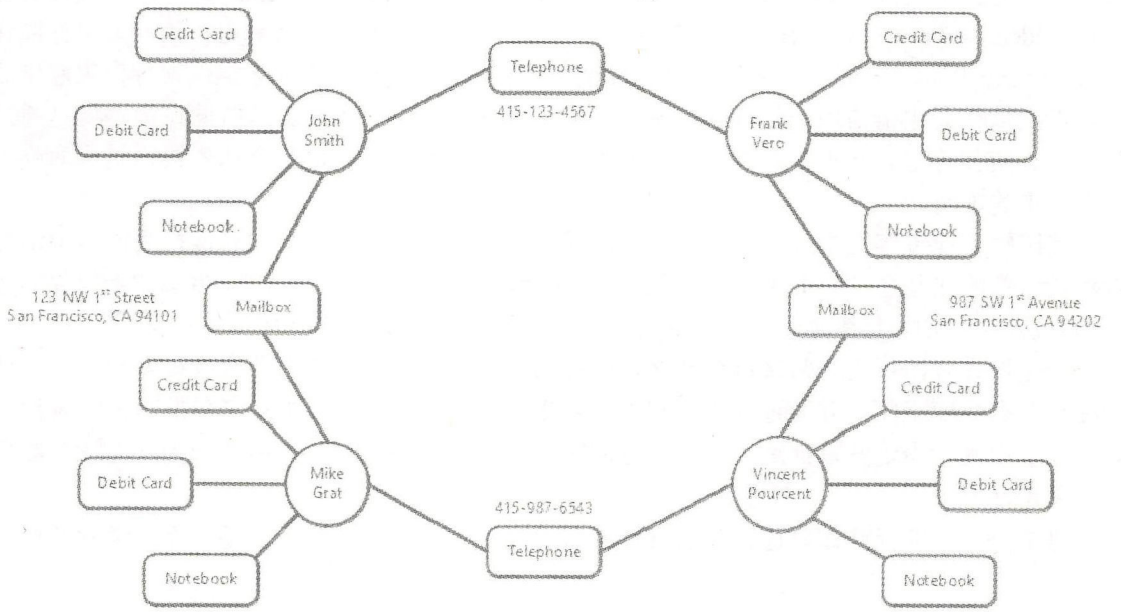


图 7-2　2 人共享 2 个数据并创建 4 个综合身份

图 7-2 显示了最终的欺诈环，每个合成身份有 4～5 个账户，总共 18 个账户。假设每个账户暴露的信用风险平均为 4,000 美元，银行的损失就可能高达 72,000 美元。在上述过程中，电话号码在环“消失”后被丢弃，当调查人员来到这些地址时，Tony Bee 和 Paul Fabre（那些真正居住在那里的欺诈者）否认知道 John Smith、Frank Vero、Mike Grat 或 Vincent Pourcent。

1. 检测罪犯

及时发现欺诈环并阻止他们造成损失是银行面临的一个挑战。传统的欺诈检测方法使用的是离散数据而没有使用连接关系，这类方法对于发现单独行动的欺诈者是有用的，但对有组织的欺诈行为却无能为力，并且这类方法也会经常出现误报情况，造成客户满意度下降并错失创造收入的机会。

Gartner 提出了一种防止欺诈的分层模型，可以从下面的图 7-3 看到：它从简单的离散方法开始（图 7-3 左侧部分），并进行到更精细的“大图”类型的分析。最右层“实体连接分析”利用连接的数据来检测有组织的欺诈。如果在客户生命周期中的关键点处使用图数据库来执行实体连接分析，可以非常容易地揭示上述类型的欺诈信息合并，具有非常高的准确率。

仅供非商业用途或交流学习使用

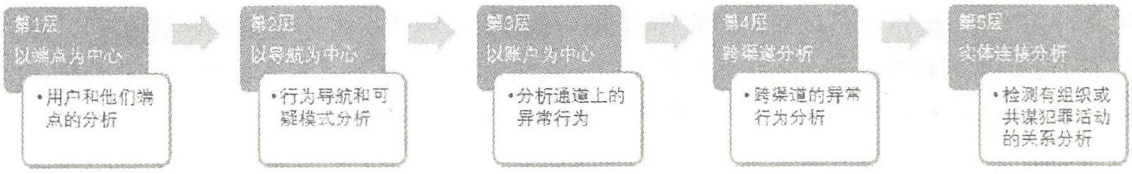


图 7-3　Gartner 的分层欺诈预防方法

如图 7-4 所示，每个共享 2 个有效标识符的 3 个人导致 9 个互连的合成身份。共享 m 个数据元素（例如姓名、出生日期、电话号码、地址、社会安全码等）的 n 个人的环（n≥2）可以创建多达 nm 个合成身份，其中对于总共(nm × m × (n-1)) / 2 个关系，每个合成身份（表示为节点）连接到 m×（n-1）个其他节点。

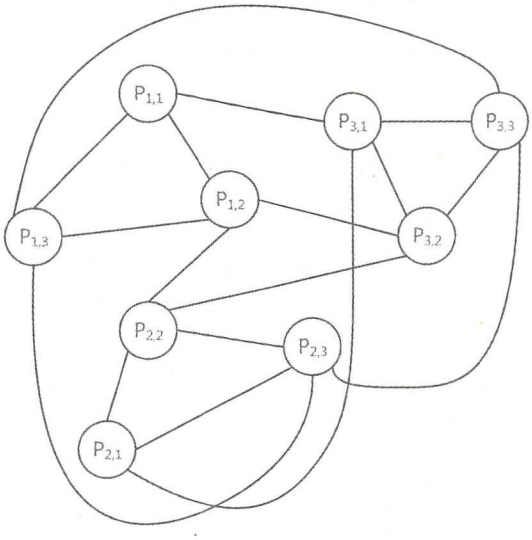


图 7-4　用 3 人合成 9 个身份

2. 实体连接分析

前面讨论了欺诈者如何使用多个身份来增加其犯罪行为的整体规模。欺诈者多重身份总数量不仅直接关系到风险敞口，也关系到发现欺诈环所需的计算复杂性。基于环的增长导致更大规模组合身份这一视角，这个问题的复杂程度就会变得越发清晰。在图 7-5 中，可以看到如何将第三个人添加到环并将扩展合成身份的数量为 9。同样，四个人可以控制 16 个身份，以此类推。假设一个 10 人欺诈环有 100 个假身份，每个假身份有 3 个金融工具，每个工具都有 5,000 美元的信用额度，总的潜在损失就是 150 万美元，这将造成一笔数额惊人的损失。

3. 图数据库解决欺诈环问题

使用传统关系数据库技术来揭露欺诈环需要将上述图形建模为一组表，然后执行一系列复杂连接和自连接。这样的查询构建起来非常复杂、运行效率低下且成本高，无法支持实时

仅供非商业用途或交流学习使用

访问，并且随着欺诈环的扩大，性能呈指数级恶化。图数据库已经成为克服这些障碍的理想工具，用 Cypher 语言可以简单地实施检测图中的欺诈环并在内存中实时导航连接。图 7-5 展示了图数据模型并且演示了通过图如何快速找到环。图数据库技术增强了现有的欺诈检测方法，并可在客户和账户生命周期的关键阶段执行并完成检测，诸如：

（1）在创建账户时。
（2）在调查期间。
（3）一旦信用余额阈值被击中。
（4）当支票退回时。

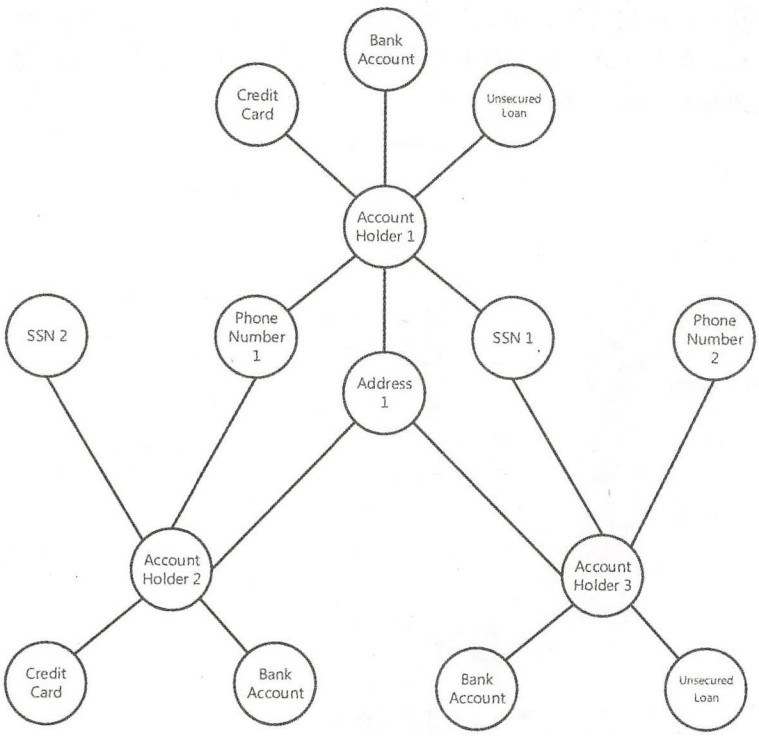


图 7-5　在图数据库中建模的欺诈环的子集

与正常类型的事件相关的实时图遍历可以帮助银行识别可能存在的欺诈环：在异常发生期间甚至在发生之前。

4. 案例数据集与程序

（1）功能：创建欺诈环。
（2）执行程序。

```
// 创建3个账户人
CREATE (accountHolder1:AccountHolder {
          FirstName: "John",
          LastName: "Doe",
          UniqueId: "JohnDoe" })
```


仅供非商业用途或交流学习使用

```
CREATE (accountHolder2:AccountHolder {
          FirstName: "Jane",
          LastName: "Appleseed",
          UniqueId: "JaneAppleseed" })
CREATE (accountHolder3:AccountHolder {
          FirstName: "Matt",
          LastName: "Smith",
          UniqueId: "MattSmith" })
// 创建地址1
CREATE (address1:Address {
          Street: "123 NW 1st Street",
          City: "San Francisco",
          State: "California",
          ZipCode: "94101" })
// 把账户人1、账户人2和账户人3关联到地址1
CREATE (accountHolder1)-[:HAS_ADDRESS]->(address1),
      (accountHolder2)-[:HAS_ADDRESS]->(address1),
      (accountHolder3)-[:HAS_ADDRESS]->(address1)
// 创建电话号码1
CREATE (phoneNumber1:PhoneNumber { PhoneNumber: "555-555-5555" })
// 把账户人1和账户人2关联到电话号码1
CREATE (accountHolder1)-[:HAS_PHONENUMBER]->(phoneNumber1),
      (accountHolder2)-[:HAS_PHONENUMBER]->(phoneNumber1)
// 创建社会安全码 SSN1
CREATE (ssn1:SSN { SSN: "241-23-1234" })
// 把账户人2和账户人3关联到 SSN1
CREATE (accountHolder2)-[:HAS_SSN]->(ssn1),
      (accountHolder3)-[:HAS_SSN]->(ssn1)
// 创建社会安全码 SSN2并关联到账户人1
CREATE (ssn2:SSN { SSN: "241-23-4567" })<-[:HAS_SSN]-(accountHolder1)
// 创建信用卡号1并关联到账户人1
CREATE (creditCard1:CreditCard {
          AccountNumber: "1234567890123456",
          Limit: 5000, Balance: 1442.23,
          ExpirationDate: "01-20",
          SecurityCode: "123" })<-[:HAS_CREDITCARD]-(accountHolder1)
// 创建银行账号并关联到账户人1
CREATE (bankAccount1:BankAccount {
          AccountNumber: "2345678901234567",
          Balance: 7054.43 })<-[:HAS_BANKACCOUNT]-(accountHolder1)
// 创建信用卡2并关联到账户人2
CREATE (creditCard2:CreditCard {
          AccountNumber: "1234567890123456",
          Limit: 4000, Balance: 2345.56,
          ExpirationDate: "02-20",
          SecurityCode: "456" })<-[:HAS_CREDITCARD]-(accountHolder2)
// 创建银行账号2并连接到账户人2
CREATE (bankAccount2:BankAccount {
          AccountNumber: "3456789012345678",
          Balance: 4231.12 })<-[:HAS_BANKACCOUNT]-(accountHolder2)
```


仅供非商业用途或交流学习使用

```
// 创建无抵押贷款2并关联到账户人2
CREATE (unsecuredLoan2:UnsecuredLoan {
          AccountNumber: "4567890123456789-0",
          Balance: 9045.53,
          APR: .0541,
          LoanAmount: 12000.00 })<-[:HAS_UNSECUREDLOAN]-(accountHolder2)
// 创建银行账号3并关联到账户人3
CREATE (bankAccount3:BankAccount {
          AccountNumber: "4567890123456789",
          Balance: 12345.45 })<-[:HAS_BANKACCOUNT]-(accountHolder3)
// 创建无抵押贷款3并关联到账号人3
CREATE (unsecuredLoan3:UnsecuredLoan {
          AccountNumber: "5678901234567890-0",
          Balance: 16341.95, APR: .0341,
          LoanAmount: 22000.00 })<-[:HAS_UNSECUREDLOAN]-(accountHolder3)
// 创建电话号码2并关联到账户人3
CREATE (phoneNumber2:PhoneNumber {
          PhoneNumber: "555-555-1234" })<-[:HAS_PHONENUMBER]-
(accountHolder3)
RETURN *
```

（3）执行结果（如图 7-6 所示）。

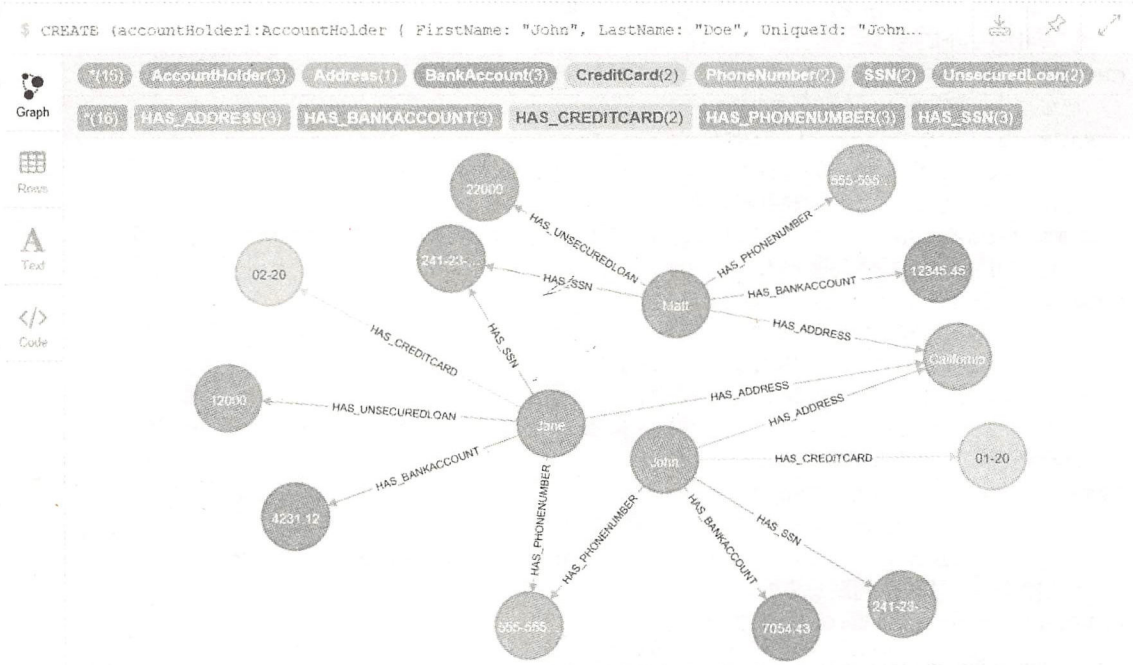


图 7-6　欺诈环案例示意图

5. 实体连接分析

下面演示对上述数据模型执行实体连接分析。在表 7-3 中使用括号是为了隔离集合的个别元素。

（1）功能：查找共享多个合法联系信息的账户持有人。

（2）执行程序。

```
MATCH       (accountHolder:AccountHolder)-[]->(contactInformation)
WITH        contactInformation,
            count(accountHolder) AS RingSize
MATCH       (contactInformation)<-[]-(accountHolder)
WITH        collect(accountHolder.UniqueId) AS AccountHolders,
            contactInformation, RingSize
WHERE       RingSize > 1
RETURN      AccountHolders AS FraudRing,
            labels(contactInformation) AS ContactType,
            RingSize
ORDER BY    RingSize DESC
```

（3）执行结果（如表 7-3 所示）。

表 7-3　查询可疑账号

欺诈环	联系类型	环大小
[MattSmith, JaneAppleseed, JohnDoe]	[Address]	3
[MattSmith, JaneAppleseed]	[SSN]	2
[JaneAppleseed, JohnDoe]	[PhoneNumber]	2

6. 确定可能的欺诈环的金融风险

（1）功能：计量欺诈风险。

（2）执行程序。

```
MATCH       (accountHolder:AccountHolder)-[]->(contactInformation)
WITH        contactInformation,
            count(accountHolder) AS RingSize
MATCH       (contactInformation)<-[]-(accountHolder),
            (accountHolder)-[r:HAS_CREDITCARD|HAS_UNSECUREDLOAN]-
>(unsecuredAccount)
WITH collect(DISTINCT accountHolder.UniqueId) AS AccountHolders,
contactInformation, RingSize,
            SUM(CASE type(r)
                WHEN 'HAS_CREDITCARD' THEN unsecuredAccount.Limit
                WHEN 'HAS_UNSECUREDLOAN' THEN unsecuredAccount.Balance
                ELSE 0
            END) as FinancialRisk
WHERE       RingSize > 1
RETURN      AccountHolders AS FraudRing,
            labels(contactInformation) AS ContactType,
            RingSize,
            round(FinancialRisk) as FinancialRisk
ORDER BY    FinancialRisk DESC
```

（3）执行结果（如表 7-4 所示）。

表 7-4　计算欺诈风险敞口

欺诈环	联系类型	环大小	金融风险
[MattSmith, JaneAppleseed, JohnDoe]	[Address]	3	34387
[MattSmith, JaneAppleseed]	[SSN]	2	29387
[JaneAppleseed, JohnDoe]	[PhoneNumber]	2	18046

7.2.2　保险欺诈

1. 案例背景

在美国，欺诈对保险业造成的损失估计每年高达 800 亿美元，这一数字近年来还一直在增长[26]。还有说法（可能有争议）指出：从 2010~2012 年，美国的可疑索赔案件上升了 27%，到 2012 年为 116,171 件，其中几乎一半来自伪造或夸大的伤害索赔[27]。在英国，保险公司估计，每个司机每年通过伪造鞭打损伤现象（Whiplash）索赔额平均增加 90 英镑[28]。这些保险欺诈均形成了复杂的犯罪环，能非常有效地规避现有的欺诈检测措施。图数据库技术再次成为打击共谋欺诈的有力工具。

2. 典型情况

以交通保险欺诈这类典型的严重欺诈情况为例，多名欺诈者一起共谋制造假交通事故并声称有人软组织损伤。显然，事故不是真实的，却有完整的假驾驶员、假乘客、假行人甚至假证人等。为什么软组织损伤这么受欺诈者所偏爱呢？主要是它容易伪造，难以验证且治疗费比较高。这个欺诈环（Fraud Ring）通常包括多种角色。

（1）协助者。各类结论或鉴定通常涉及以下几类专业人士的参与：

- 医生，开具诊断虚假受伤证明或鉴定。
- 律师，提出欺诈索赔。
- 汽车实体店，谎报损坏的汽车。

（2）参加者。指涉及假事故的人，通常包括：

- 司机。
- 乘客。
- 行人。
- 证人。

欺诈者通常采用参加者角色的轮换来创建和管理欺诈环，以便制造大量的假事故，也就是说，不同事故中参加者的角色可能不断重复改变，从而确保即便只有几名参加者也可以制造大量假事故。如图 7-7 所示，6 人共谋了 3 起假事故。每个人扮演“司机”一次、“乘

[26] http://www.insurancefraud.org/article.htm?RecID=3274#.UnWuZ5E7ROA
[27] https://www.nicb.org/newsroom/news-releases/u-s--questionable-claims-report
[28] http://www.insurancefraud.org/IFNS-detail.htm?key=17499#.UmmsJyQhZ0o

客”两次。假设平均每个受伤人员索赔 20,000 美元，每辆车 5,000 美元，那么这个仅 6 人构成的欺诈环将能索赔 390,000 美元。

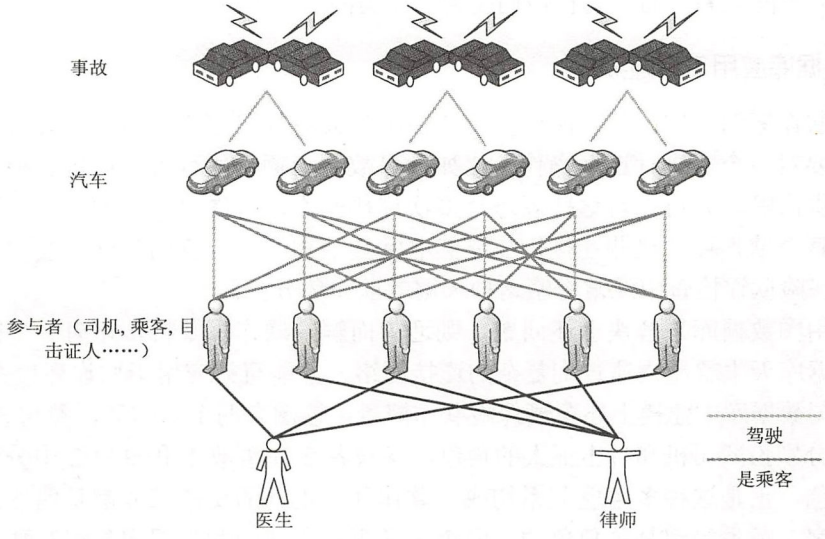


图 7-7　六人共谋

与“7.2.1 银行欺诈”小节介绍的银行欺诈案例类似，这种合谋诈骗的复杂性和规模可以快速上升。比如，在 10 个人合谋进行保险欺诈的例子中，假定上演 5 起假事故，如图 7-8 所示，其中每个人扮演司机和证人各一次、乘客三次，每个受伤人员的平均索赔额为 4 万美元，每辆车的赔偿额为 5 万美元，那么这个欺诈环可以制造出 40 人受伤，索赔高达 160 万美元！

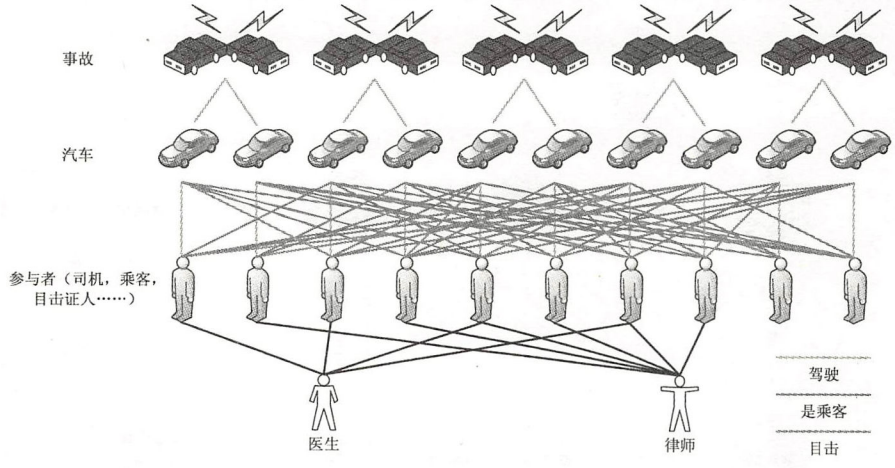


图 7-8　10 人共谋

3. 检测犯罪

与银行欺诈检测类似，分层方法已经成为检测保险欺诈的主要手段。虽然现有的分析技术足以检测出某些特定的欺诈行为，但是富有经验的犯罪分子通常通过共谋欺骗能有效地避

开这些检测手段，而且不容易被引起怀疑。

下一代保险欺诈检测技术是使用社交网络分析来发现这些欺诈环。通过连接分析能发现隐藏的人与人之间关系，而不会将他们视为“完美的陌生人”。

4. 图数据库应用于保险欺诈

与银行欺诈案例一样，社交网络分析往往不是关系数据库的强项。在关系数据库中，发现欺诈环需要对多个表进行连接操作，比如：事故、车辆、所有者、司机、乘客、行人、目击证人、提供商等，并可能对这些表进行多次连接操作，每次关联以发现一个潜在角色，从而最终获取整个欺诈环。这些连接操作显然非常复杂和低效，特别是针对非常大的数据集，这将导致在保险欺诈检查中经常忽略这种关键的欺诈环分析。

如果采用图数据库来解决上述问题，则迎刃而解，欺诈报警将转化为一个简单的图遍历问题。图数据库天生就能快速查询复杂的连接网络，并能直接应用于欺诈环检查。图 7-9 描述了如何在图数据库中建模上述车辆保险欺诈情景：该图有两个欺诈环，参与者 1 在事故 1 和事故 2 中分别扮演司机和目击证人的角色，参与者 2 在事故 1 和事故 2 中分别扮演了司机和乘客的角色，正是这种多角色关系构成了欺诈环。正常的交通事故都是偶然的，司机、目击证人、乘客、医生等都是单角色的，两个正常事故之间的关系图谱是相互独立的，形成环的概率非常低。假交通事故中的职业欺诈者为了提高欺诈收益并降低成本，会多次轮换角色“制造”若干交通事故，这必然形成欺诈环。

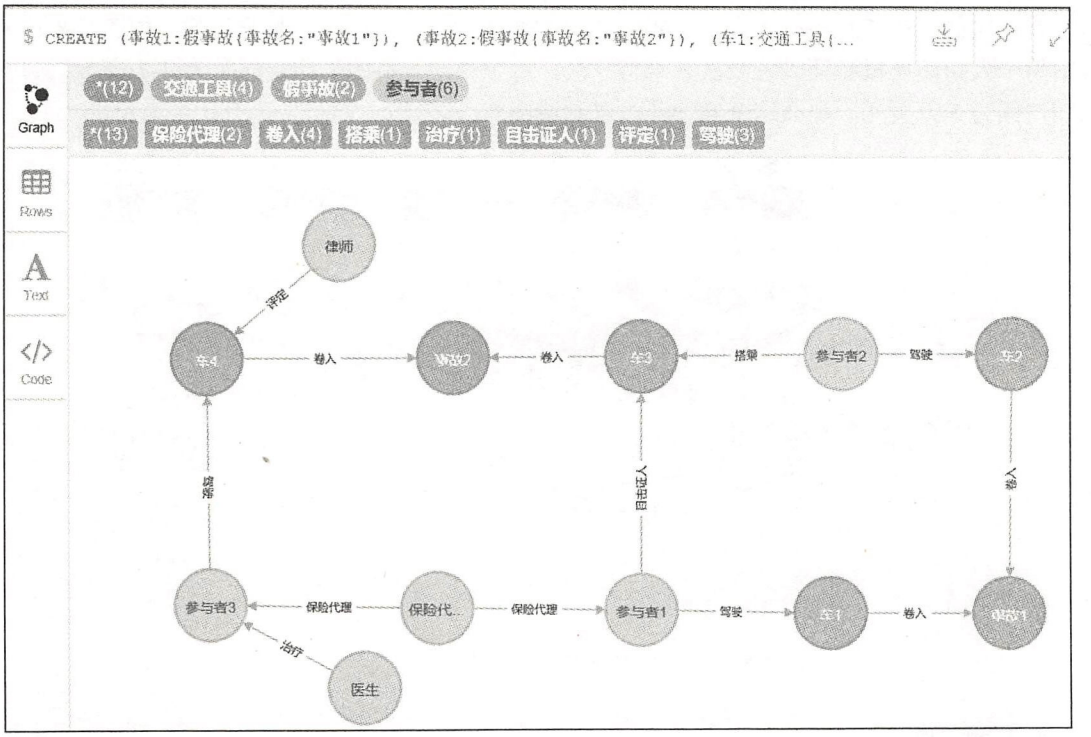


图 7-9　保险欺诈的图表示

与“7.2.1 银行欺诈”小节的银行欺诈案例类似，可以在图数据库中添加相应的检查，触

发检查的时间点比如在提交索赔时，就能实时标记出疑似欺诈的行为。图 7-9 的创建程序代码如下：

```
CREATE
(事故 1:假事故{事故名:"事故 1"}),
(事故 2:假事故{事故名:"事故 2"}),
(车 1:交通工具{作者名:"车 1"}),
(车 2:交通工具{作者名:"车 2"}),
(车 3:交通工具{作者名:"车 3"}),
(车 4:交通工具{作者名:"车 4"}),
(参与者 1:参与者{名称:"参与者 1"}),
(参与者 2:参与者{名称:"参与者 2"}),
(参与者 3:参与者{名称:"参与者 3"}),
(保险代理人:参与者{名称:"保险代理人"}),
(医生:参与者{名称:"医生"}),
(律师:参与者{名称:"律师"}),
(车 1)-[:卷入]->(事故 1),
(车 2)-[:卷入]->(事故 1),
(车 3)-[:卷入]->(事故 2),
(车 4)-[:卷入]->(事故 2),
(参与者 1)-[:驾驶]->(车 1),
(参与者 2)-[:驾驶]->(车 2),
(参与者 3)-[:驾驶]->(车 4),
(参与者 1)-[:目击证人]->(车 3),
(参与者 2)-[:搭乘]->(车 3),
(律师)-[:评定]->(车 4),
(保险代理人)-[:保险代理]->(参与者 1),
(保险代理人)-[:保险代理]->(参与者 3),
(医生)-[:治疗]->(参与者 3)
RETURN *
```

7.2.3　电子商务欺诈

1. 案例背景

随着网络和信息技术的高速发展，现代生活越来越离不开网络，大量的金融交易在线上完成。欺诈者同样与时俱进，并设计出更为聪明的方法来欺骗在线支付系统。虽然这类行为可以而且确实涉及犯罪环（Criminal Ring），但高智商欺诈者能够虚假构造出大量的合成身份，并使用这些身份来完成相当大的欺诈计划。

2. 典型情况

假定一个在线交易系统，有以下标识符：用户 ID、IP 地址、地理位置、可跟踪的 cookie 和信用卡号。我们通常期望这些标识符之间是一一对应的。当然，有些标识可能是共享的，比如：一个家庭共享一个信用卡号，一台电脑多个人使用等。但是，某些关系超过合理的数值，则极有可能是欺诈。例如，同一个 IP 上大量用户进行交易，同一张信用卡向不同的地址大量供货，或多张信用卡使用同一个地址等。这些问题完全可以通过遍历不同信息之间的关系来发现图中的欺诈环，并将关联的程度作为一个强指示信号。标识符之间联系越多，发生欺诈的概率就越大，大而紧密的图是欺诈正在发生的非常强的指示信号。

3. 图数据库改进电子商务欺诈检测

与银行欺诈和保险欺诈类似，图数据库能实时有效地从这些数据中检测出电子商务欺诈。通过预置相关检测机制并与相应事件触发器相关联，事件触发器可以包括诸如登录、下订单或注册信用卡等事件，这样就能及早地检测出电子商务欺诈。图 7-10 中表示来自不同 IP 地址的一系列商务行为，其中 IP_1 很可能是欺诈地址源。

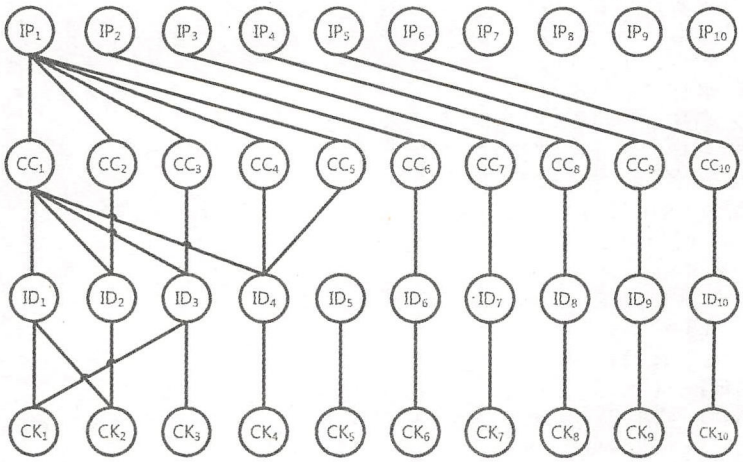


图 7-10　来自地址 IP_1 的在线付款欺诈

上图中 IP_x 表示不同的 IP 地址，CC_x 表示不同的信用卡号，ID_x 表示不同的在线交易用户，CK_x 表示存储在系统中的特定 cookie。在示例中，一个 IP 使用五张信用卡进行多次交易，其中一张信用卡 CC_1 由多个用户 ID 使用，其中 CK_1 和 CK_2 每个共享两个用户 ID。

7.2.4　小结

无论是银行欺诈、保险欺诈、电子商务欺诈还是其他类型的欺诈，有两点是很明确的：一是要尽快发现欺诈，以便尽早终止，降低危害——随着业务流程更便捷和自动化，用于检测欺诈的时间将大大缩短，无疑对欺诈检测的时效性提出了更高的要求；二是重视关系分析，有经验的罪犯能够有效地攻击安全防护比较弱的系统，传统技术虽仍能预防某些欺诈类型，但对复杂的欺诈环检测将无能为力。这正是图数据库可以发挥其价值的地方。

图数据库是欺诈检测的理想工具。无论是欺诈环、合谋团体，还是高智商犯罪分子，图数据库都具备实时发现各种复杂欺诈模式的独特能力，从而使欺诈无处藏身。

7.3　科研导图

本节主要介绍 Neo4j 在科学研究和技术创新领域的一些典型应用。

根据中国科学技术发展战略研究院发布的《国家创新指数报告 2015》[29]显示，2014 年中

[29] http://www.most.gov.cn/kjbgz/201607/t20160726_126754.htm

国研究与开发（Research and Development，R&D）经费为 2,118.6 亿美元，位居世界第 2 位，占全球份额的 14.4%，2000～2014 年中国 R&D 经费年均增速为 16.4%，位居世界首位，大幅领先其他国家，中国 R&D 经费占到 GDP 的 2.05%，超越欧盟 28 国总体水平（1.94%）。2014 年，中国 R&D 人员总量为 371.1 万人，连续 8 年位居世界首位，占全球 R&D 人员总量的 31.3%。2014 年，中国 SCI 论文数量达到 25 万篇，连续 7 年位居世界第 2 位，中国 SCI 论文数量占到全球总量的 13.3%。2014 年在中国，国内发明专利申请量达到 80.1 万件，占世界总量的 47.5%，连续 5 年居世界首位，中国国内发明专利授权量达 16.3 万件，占世界总量的 24.8%，连续 4 年位居世界第 2 位，与第 1 位的日本差距进一步缩小。2016 年，我国 R&D 经费支出 1.55 万亿元人民币，同比增长 9.4%，占国内生产总值的 2.08%，共受理境内外专利申请 346.5 万件，授予专利权 175.4 万件[30]。2016 年，SCI 收录中国学者发表的论文共 30.4 万篇[31]。由此可见，我国已经是科研大国，科研成果取得的经济效益在国民经济中已经占到比较重要的位置。

但我国科研界也存在明显的痛点问题。这里仅指出几个与科研行为和科研成果相关的典型问题。一是重复科研问题，限于能力、懒惰或科研条件制约等原因，科研人员未能对前人相关成果进行全面整理，结果导致重复科研，造成资源浪费。二是科研资料搜索技术智能水平较低，当前的科研搜索仍然处于科研资源数据库搜索、谷歌学术搜索、百度学术搜索等互联网搜索技术阶段，这种搜索仅是把相关的搜索结果以列表形式按照某种指标进行排序，这种模式并不能揭示搜索结果之间的结构或关系，而这种结构或关系在科研活动中往往更有价值。三是科研社交问题，当前我国科研社交仍然处于初级阶段，主要特征是细分科研领域的封闭性，不同细分领域很少有来往，科研机构与企业关联少，导致企业不能及时找到需要的专家，科研机构也不能有效地把科研成果拓展到企业或其他交叉领域。交叉创新是未来科研的主要创新方式之一，未来需要一种新型的科研社交平台。四是科研成果抄袭问题，抄袭或剽窃他人科研成果在我国仍会经常发生，基于现有的学术成果查重平台，这个问题已经得到了一定程度上的遏制。

Neo4j 无疑是解决上述提及的科研痛点问题的有力工具。本节将基于互联网上中国知网和万方网站公开可获取的论文作者、论文题目、作者单位、论文关键词、论文摘要、论文参考文献等数据——图 7-11 所示就是描述这些数据的关系图谱，构建学术科研导图应用平台。该平台可以实现以下功能：

- 构建科研社交网络：科研人员可以通过科研成果的关键词进行归类，并形成科研人员的网络关系。此时，关键词是科研人员的属性。
- 专家推荐：根据科研成果数据，通过 Neo4j 的计算获得某个领域专家的实力排名。
- 文献快速搜索和整理：通过 Neo4j 可以快速实现相关文献的快速搜索并以图的形式展示，而不仅仅是科研成果的简单列表，这有利于整理文献综述，而且不会出现把重要的文献漏掉的情况。
- 合作关系图谱：利用 Neo4j 可以很方便计算任意两个单位之间的合作程度，或任意两位科研人员之间的合作程度。

[30] http://www.stats.gov.cn/tjsj/zxfb./201702/t20170228_1467424.html
[31] http://www.letpub.com.cn/index.php?page=sci_colleges_rank_2016

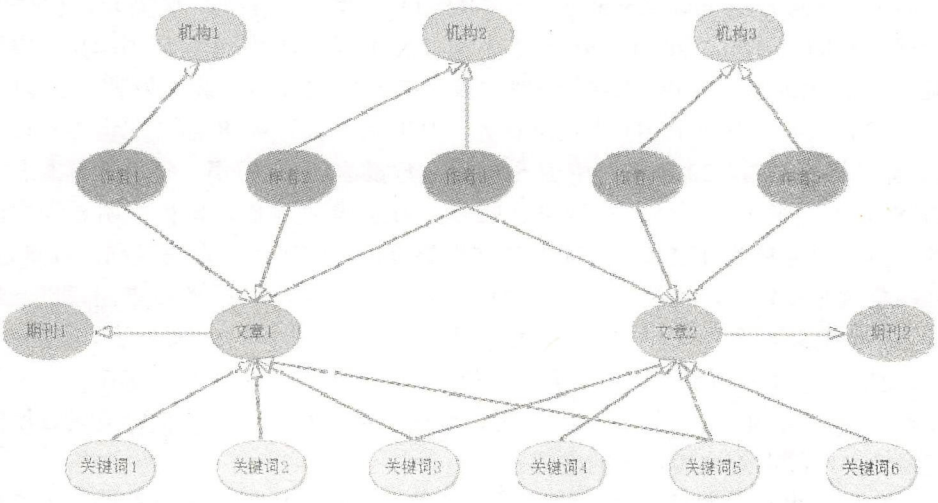


图 7-11　科研图谱

1. 专家推荐

需求描述。在科研人员搜索某个研究领域的文献或查找该领域相关专家的时候，快速而准确地给出该领域有重要影响力的专家，将会极大方便科研人员掌握该领域目前的研究状况。查找权威专家的传统方法是根据专家作品的影响力来决定，也就是基于统计的方法，这个方法简单而且适用范围比较广，但是却忽略了专家之间的关系，也就是专家之间的合作网络。

如果用图数据库来存储这些数据就可以有效地解决这样的问题。利用图数据库不仅可以通过专家的作品影响力来推荐，还可以利用专家在图中的位置以及位置的重要性来推荐。通过求出每个专家在图中的介数中心性（Betweenness Centrality）、紧密度中心性（Closeness Centrality）、度中心性（Degree Centrality）和特征向量中心性（PageRank Centrality），然后就可以根据不同的需求推荐合适的专家，如图 7-12 所示。

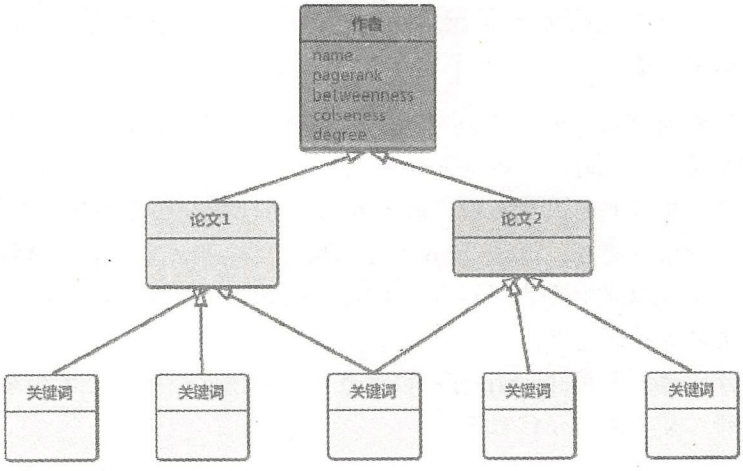


图 7-12　专家影响力计算示意图

2. 创建科研导图示范案例

（1）功能：本案例用于描述研究人员、论文、关键词、学术期刊等之间的关系。

（2）执行程序。

```
CREATE
(张教授:研究人员{姓名:"张教授"}),
(李老师:研究人员{姓名:"李老师"}),
(王老师:研究人员{姓名:"王老师"}),
(论文 1:论文{论文名:"论文 1"}),
(论文 2:论文{论文名:"论文 2"}),
(论文 3:论文{论文名:"论文 3"}),
(论文 4:论文{论文名:"论文 4"}),
(论文 5:论文{论文名:"论文 5"}),
(论文 6:论文{论文名:"论文 6"}),
(论文 7:论文{论文名:"论文 7"}),
(论文 8:论文{论文名:"论文 8"}),
(学术期刊 1:期刊{刊名:"学术期刊 1"}),
(学术期刊 2:期刊{刊名:"学术期刊 2"}),
(学术期刊 3:期刊{刊名:"学术期刊 3"}),
(关键词 1_1:关键词{词名:"关键词 1_1"}),
(关键词 1_2:关键词{词名:"关键词 1_2"}),
(关键词 1_3:关键词{词名:"关键词 1_3"}),
(关键词 2_1:关键词{词名:"关键词 2_1"}),
(关键词 2_2:关键词{词名:"关键词 2_2"}),
(关键词 2_3:关键词{词名:"关键词 2_3"}),
(关键词 3_1:关键词{词名:"关键词 3_1"}),
(关键词 3_2:关键词{词名:"关键词 3_2"}),
(关键词 3_3:关键词{词名:"关键词 3_3"}),
(张教授)-[:作者]->(论文 1),
(张教授)-[:作者]->(论文 2),
(张教授)-[:作者]->(论文 3),
(张教授)-[:作者]->(论文 4),
(张教授)-[:作者]->(论文 5),
(张教授)-[:作者]->(论文 6),
(张教授)-[:作者]->(论文 7),
(张教授)-[:作者]->(论文 8),
(李老师)-[:作者]->(论文 1),
(李老师)-[:作者]->(论文 2),
(王老师)-[:作者]->(论文 5),
(王老师)-[:作者]->(论文 6),
(论文 1)-[:发表]->(学术期刊 1),
(论文 2)-[:发表]->(学术期刊 1),
(论文 3)-[:发表]->(学术期刊 1),
(论文 4)-[:发表]->(学术期刊 2),
(论文 5)-[:发表]->(学术期刊 2),
(论文 6)-[:发表]->(学术期刊 2),
(论文 7)-[:发表]->(学术期刊 3),
(论文 8)-[:发表]->(学术期刊 3),
(论文 1)-[:关键词]->( 关键词 1_1),
(论文 1)-[:关键词]->( 关键词 1_2),
(论文 1)-[:关键词]->( 关键词 1_3),
(论文 2)-[:关键词]->( 关键词 2_1),
(论文 2)-[:关键词]->( 关键词 2_2),
(论文 2)-[:关键词]->( 关键词 2_3),
```

```
(论文 3)-[:关键词]->( 关键词 2_3),
(论文 4)-[:关键词]->( 关键词 1_2),
(论文 5)-[:关键词]->( 关键词 3_1),
(论文 5)-[:关键词]->( 关键词 3_2),
(论文 5)-[:关键词]->( 关键词 3_3),
(论文 6)-[:关键词]->( 关键词 2_1),
(论文 7)-[:关键词]->( 关键词 3_1),
(论文 8)-[:关键词]->( 关键词 3_2)
RETURN *
```

（3）执行结果（图 7-13 所示）。

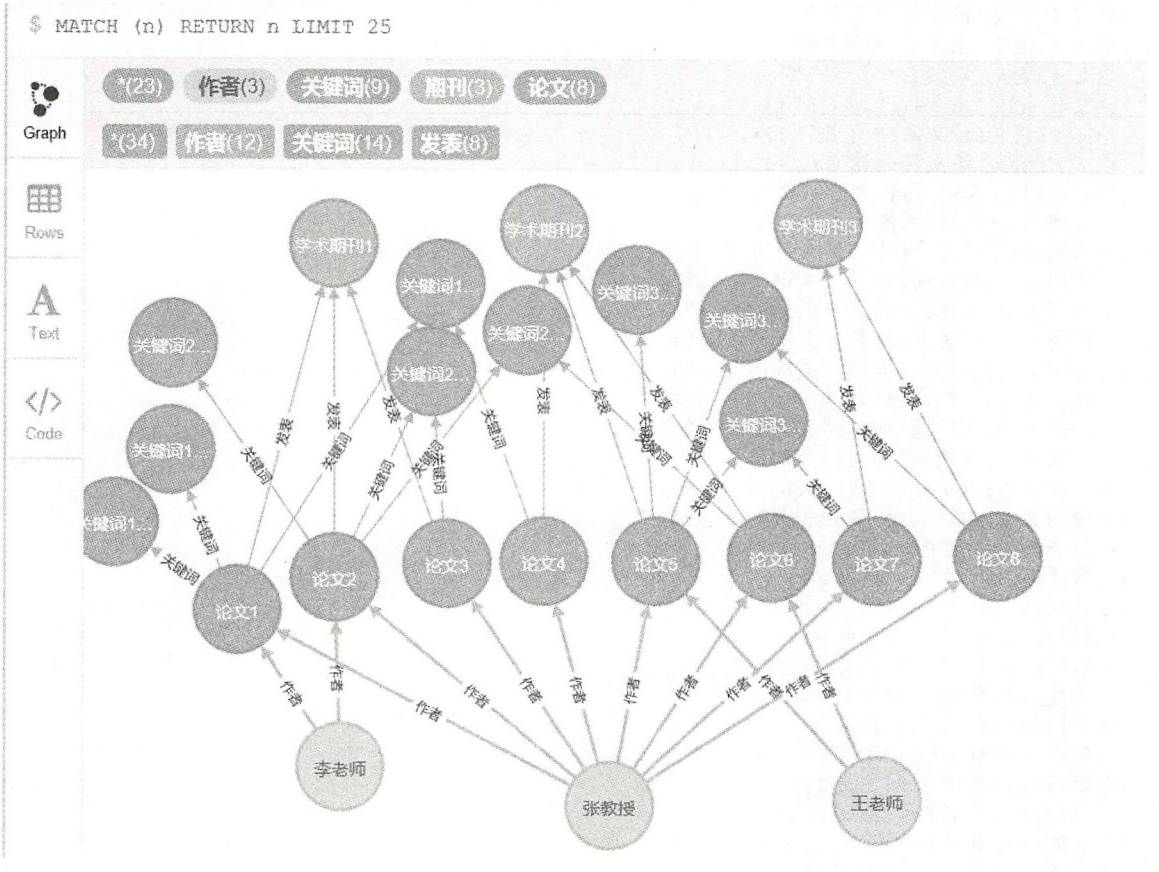


图 7-13　学术成果关系图谱

（4）汇总计算三位作者所发表的文章数量。

执行下面的程序：

```
MATCH (p:研究人员)-[a:作者]->() RETURN p.姓名 AS 作者姓名,
COUNT(a) AS 发表论文数
```

程序执行结果（如表 7-5 所示）。

表 7-5　统计作者文章数量

作者姓名	发表论文数
张教授	8
王老师	2
李老师	2

3. 文献搜索

（1）需求描述

科研文献检索是每个科研工作者在日常科研工作中常常要面临的问题，但是现存的检索系统往往采用的是全文检索，全文检索的好处是精确度较高，可是也存在着效率较低，并且可靠性关联度不高的问题。

如果使用 Neo4j 进行检索，不仅可以快速地检索出匹配度最高的论文，还可以把关联度较高的论文也一同检索出来，极大地丰富了检索内容的多样性，如图 7-14 所示。

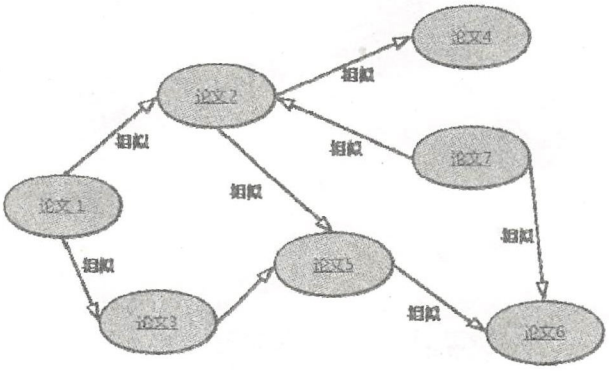


图 7-14　查找相似论文路径示意图

首先利用全文检索出最相似的论文，再根据查找出来的论文寻找与它具有“相似”关系的论文即为相似论文。确立目标论文，然后通过计算找到相似的论文，建立相似关系，用 Neo4j 把这种相似性表达出来。

（2）功能：创建相似论文图谱。

（3）执行程序。

```
CREATE
(论文 1:论文图谱{论文名:"论文 1"}),
(论文 2:论文图谱{论文名:"论文 2"}),
(论文 3:论文图谱{论文名:"论文 3"}),
(论文 4:论文图谱{论文名:"论文 4"}),
(论文 5:论文图谱{论文名:"论文 5"}),
(论文 6:论文图谱{论文名:"论文 6"}),
(论文 7:论文图谱{论文名:"论文 7"}),
(论文 1)-[:相似]->(论文 2),
(论文 1)-[:相似]->(论文 3),
(论文 2)-[:相似]->(论文 4),
```

```
(论文 2)-[:相似]->(论文 5),
(论文 3)-[:相似]->(论文 5),
(论文 5)-[:相似]->(论文 6),
(论文 7)-[:相似]->(论文 2),
(论文 7)-[:相似]->(论文 6)
RETURN *
```

（4）执行结果（如图 7-15 所示）。

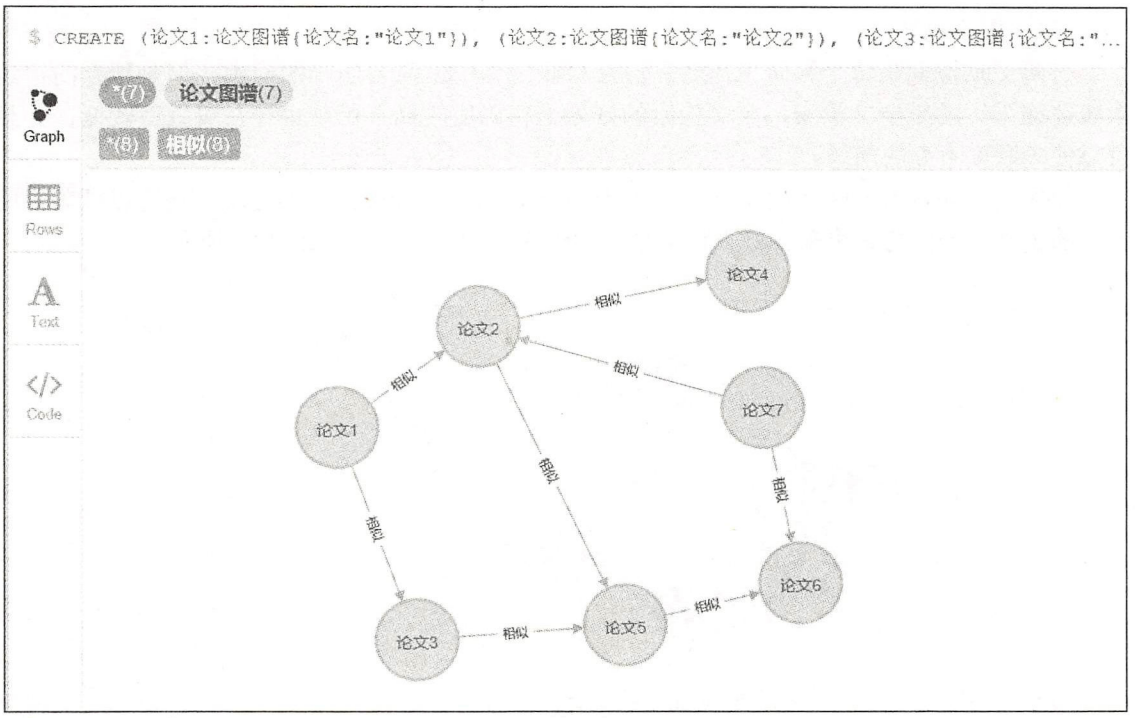


图 7-15　相似论文图谱

（5）找出论文 1 与论文 6 这两篇论文之间相似传递的路径。

执行程序：

```
MATCH n=allshortestPaths((论文1:论文图谱{论文名:"论文1"})-[*..6]-> (论文6:论文图谱
{论文名:"论文6"}))
RETURN n
```

执行结果（如图 7-16 所示）。

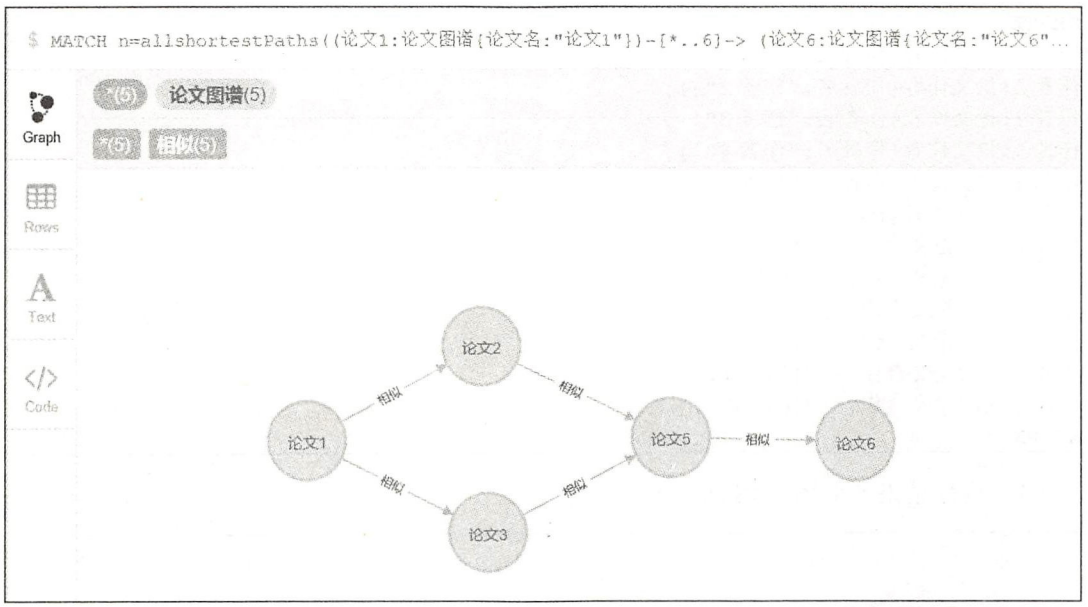


图 7-16　两篇相似论文的所有最短路径

4. 科研合作图谱

（1）需求描述

在科研导图的分析中还有一个重要的工作，那就是研究不同科研人员之间的合作和不同科研机构之间的合作。当某个科研人员或者是某个科研机构有意向寻找合作伙伴时，合作图谱可以起到较大的辅助作用，科研合作示意如图 7-17 所示。

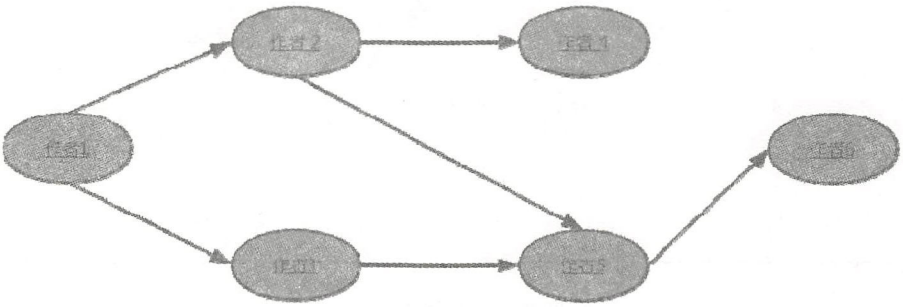


图 7-17　科研合作社交示意图

假设作者 1 想和作者 5 进行合作，那么首先查找作者 1 和作者 5 之间的最短路径，可以找到两条最短路径为作者 1→作者 2→作者 5 和作者→作者 3→作者 5，其次通过作者 3 或作者 2 的介绍可以让作者 1 和作者 5 能够相互认识。

（2）功能：创建科研合作图谱。

（3）执行程序。

```
CREATE
(作者1:论文作者{作者名:"作者1"}),
(作者2:论文作者{作者名:"作者2"}),
(作者3:论文作者{作者名:"作者3"}),
(作者4:论文作者{作者名:"作者4"}),
(作者5:论文作者{作者名:"作者5"}),
(作者6:论文作者{作者名:"作者6"}),
(作者1)-[:论文合作]->(作者2),
(作者1)-[:论文合作]->(作者3),
(作者2)-[:论文合作]->(作者4),
(作者2)-[:论文合作]->(作者5),
(作者3)-[:论文合作]->(作者5),
(作者5)-[:论文合作]->(作者6)
RETURN *
```

（4）执行结果（如图 7-18 所示）.

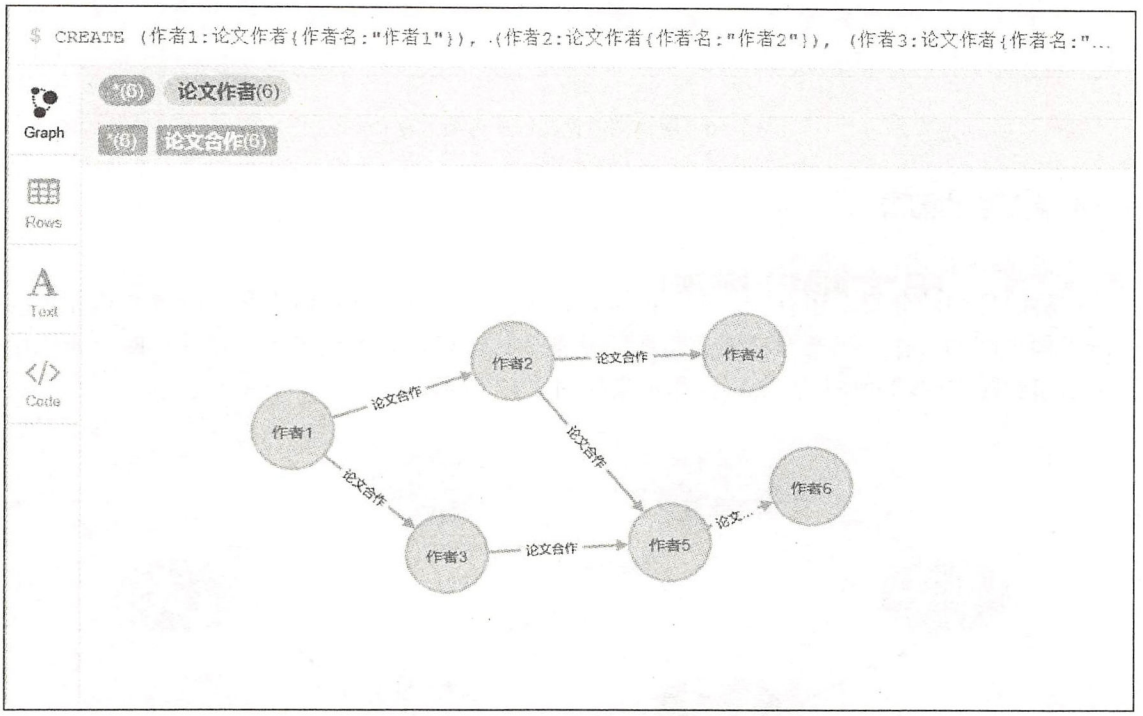


图 7-18　科研合作图谱

（5）查找两位作者的最短路径。

执行程序：

```
MATCH n=allshortestPaths((作者1:论文作者{作者名:"作者1"})-[*..6]-(作者1:论文作者
{作者名:"作者5"}))
RETURN n
```

执行结果（如图 7-19 所示）。

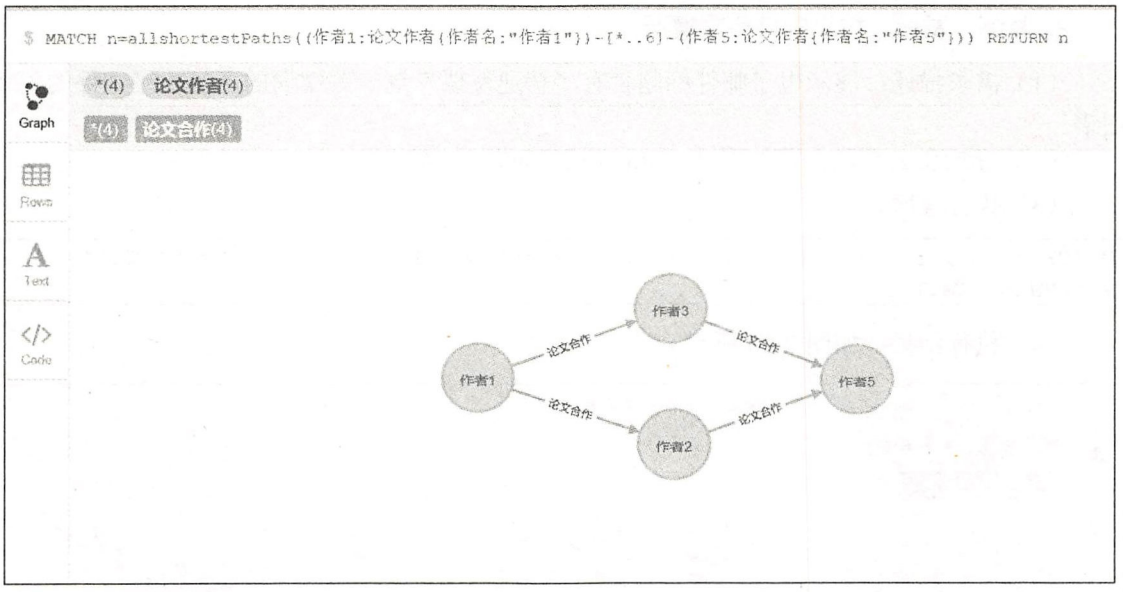


图 7-19　最短科研社交路径

7.4 电子邮件监测

本节重点介绍 Neo4j 在电子邮件监测领域的一些典型应用。

问题描述：鉴于互联网泄密事件频发，电子邮箱地址已经批量流入数据黑市。存在一些机构从数据黑市批量采购这些电子邮箱，然后批量发送垃圾电子邮件甚至非法电子邮件，给用户带来很大烦恼，每个用户都会频繁收到大量垃圾邮件，以至于邮箱的价值已经大打折扣。对各种垃圾电子邮件或违法行为进行及时的监控、追踪，必要时对犯罪嫌疑人采取抓捕行动，成为互联网安全的基本内容之一。

解决方案：Neo4j 是解决这一问题的有力工具。把发送人和接收人的电子邮箱地址、邮件标题、发送时间等数据收集整理，并导入 Neo4j 形成电子邮件图数据库。基于该电子邮件图数据库，可以实现以下功能：

- 标题含有某个关键词的邮件图谱展示：输入某个关键词，与该关键词有关的邮件全部以图的形式自动展现。
- 识别发送垃圾邮件或非法邮件的重点监测对象：设定一个阈值，一旦发送某种垃圾电子邮件或非法电子邮件达到某个数量，就认定为重点监测对象。

1. 数据与案例

本节收集了 50 万封电子邮件。数据项包括发送人和接收人的电子邮箱地址、邮件标题、发送时间等。

2. 监测“发票”垃圾邮件传播情况

（1）需求描述：展示电子邮件标题含有“普通发票”这一关键词的所有电子邮件的关系图谱。

（2）登录微云数聚公司的网站：http://we-yun.com:7474/。

（3）执行程序。

```
MATCH m=(s:Person)-->(e:Email)-->(r:Person) WHERE e.title=~'.*普通发票.*'
RETURN m LIMIT 15
```

（4）执行结果（如图 7-20 所示）。

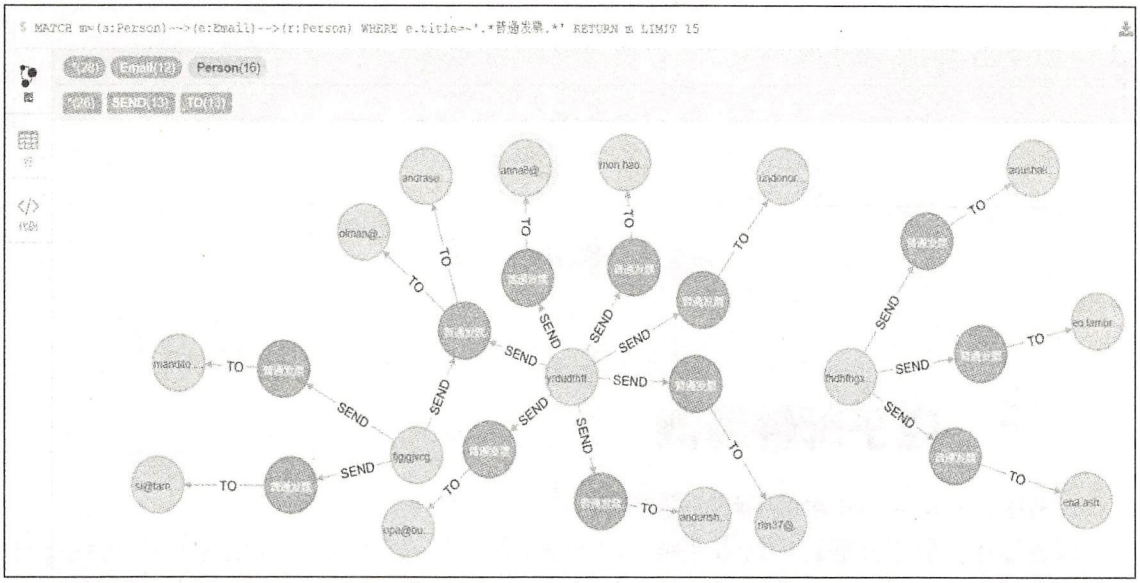


图 7-20　显示邮件标题包含“普通发票”的邮件图谱

3. 案例 2——寻找“主犯”

（1）需求描述：寻找“主犯”——展示电子邮件标题含有“发票”这一关键词且发送量超过 105 份的所有电子邮件的关系图谱。

（2）登录微云数聚公司的网站：http://we-yun.com:7474/。

（3）执行程序。

```
MATCH(s:Person)-->(e:Email)-->(r:Person)
WHERE e.title=~'.*发票.*'
WITH s,COUNT(e) AS num,COLLECT(e) AS emails, COLLECT(r) AS receives
WHERE num >105
RETURN s,emails,receives
```

（4）执行结果（如图 7-21 所示）。

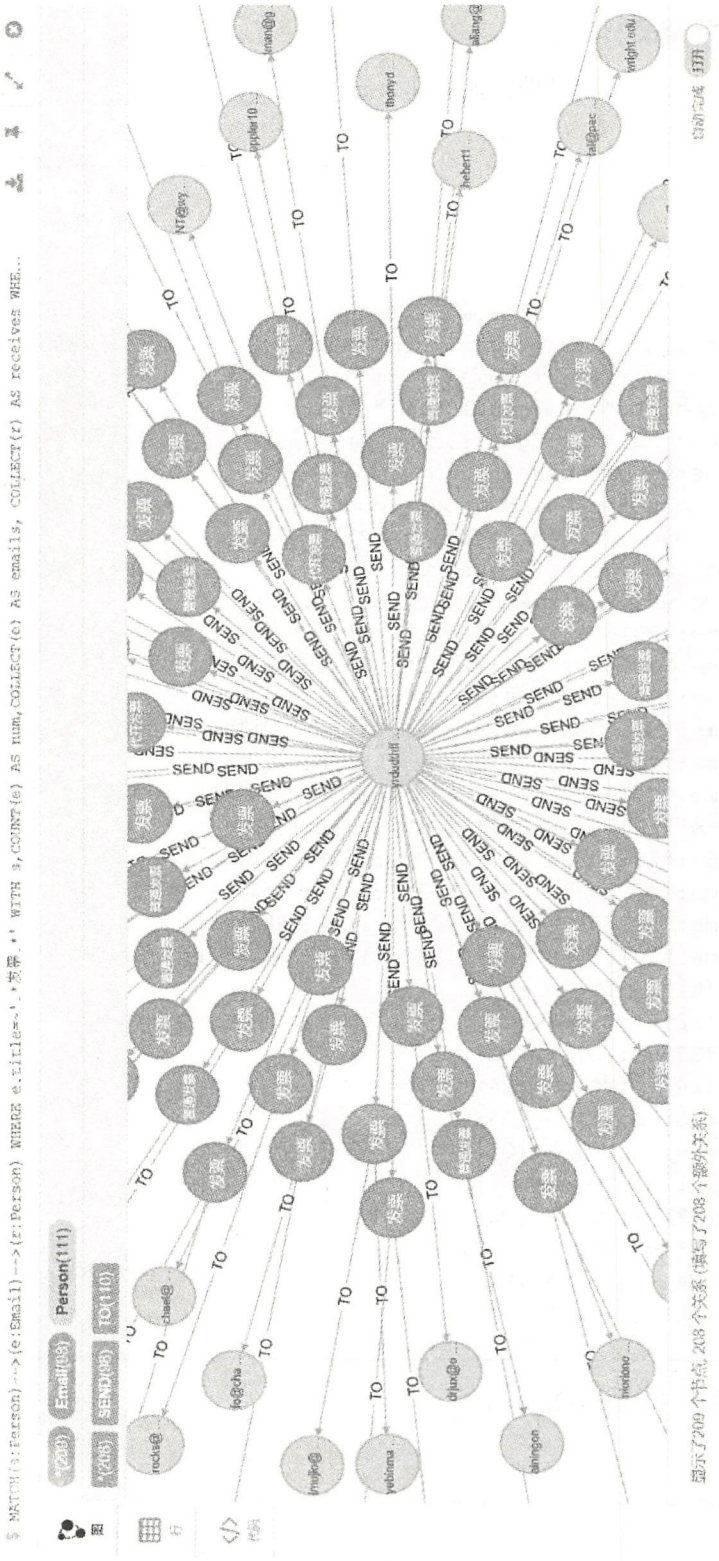


图 7-21　查找垃圾邮件“主犯”

4. 邮件检测示范案例

（1）功能：实现垃圾邮件监测的创建。

（2）执行程序。

```
CREATE
(A:Person{account:"fhdhfhgxcvvxg@163.com"}),
(B:Person{account:"yrdudthfffxddh@126.com"}),
(C1: Person{account:"rbs0901@gmail.com"}),
(C2: Person{account:"eo.lambro@gmail.com "}),
(C3: Person{account:"i.rao@gmail.com "}),
(C4: Person{account:"yuanli@gmail.com "}),
(C5: Person{account:"hoke@gmail.com "}),
(C6: Person{account:"zeta@gmail.com "}),
(D1: Person{account:"uida@sinoclc.com"}),
(D2: Person{account:"uwafir@sinoclc.com"}),
(D3: Person{account:"areg@sinoclc.com"}),
(D4: Person{account:"intern11@sinoclc.com"}),
(D5: Person{account:"liduo@sinoclc.com"}),
(E1: Email{time:"20160802 04:58:01",title:"发票"}),
(E2: Email{time:"20160802 04:58:02",title:"发票"}),
(E3: Email{time:"20160802 04:58:03",title:"发票"}),
(E4: Email{time:"20160802 04:58:04",title:"发票"}),
(E5: Email{time:"20160802 04:58:05",title:"发票"}),
(E6: Email{time:"20160802 04:58:06",title:"发票"}),
(F1: Email{time:"20170308 05:15:01",title:"招聘"}),
(F2: Email{time:"20170308 05:15:02",title:"招聘"}),
(F3: Email{time:"20170308 05:15:03",title:"招聘"}),
(F4: Email{time:"20170308 05:15:04",title:"招聘"}),
(F5: Email{time:"20170308 05:15:05",title:"招聘"}),
(A)-[:SEND]->(E1)-[:TO]-> (C1),
(A)-[:SEND]->(E2)-[:TO]-> (C2),
(A)-[:SEND]->(E3)-[:TO]-> (C3),
(A)-[:SEND]->(E4)-[:TO]-> (C4),
(A)-[:SEND]->(E5)-[:TO]-> (C5),
(A)-[:SEND]->(E6)-[:TO]-> (C6),
(B)-[:SEND]->(F1)-[:TO]-> (D1),
(B)-[:SEND]->(F2)-[:TO]-> (D2),
(B)-[:SEND]->(F3)-[:TO]-> (D3),
(B)-[:SEND]->(F4)-[:TO]-> (D4),
(B)-[:SEND]->(F5)-[:TO]-> (D5)
RETURN *
```

（3）执行结果（如图 7-22 所示）。

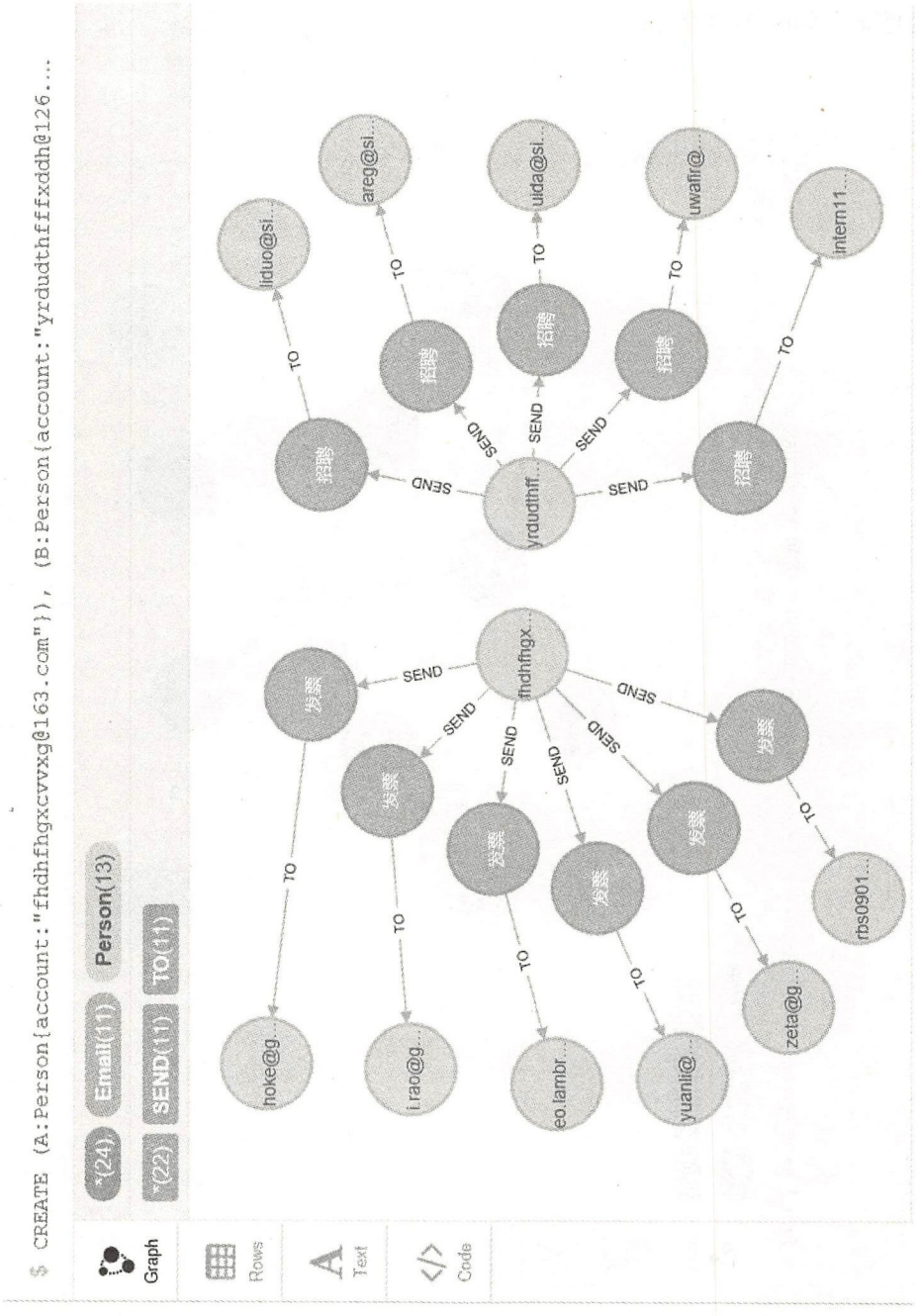


图 7-22　垃圾邮件监测示意图

（4）监测邮件标题含有“招聘”的发件人、邮件、收件人及其关系。

执行程序：

```
MATCH m=(s:Person)-->(e:Email)-->(r:Person)
WHERE e.title=~".*招聘.*"
RETURN m
```

执行结果（如图 7-23 所示）。

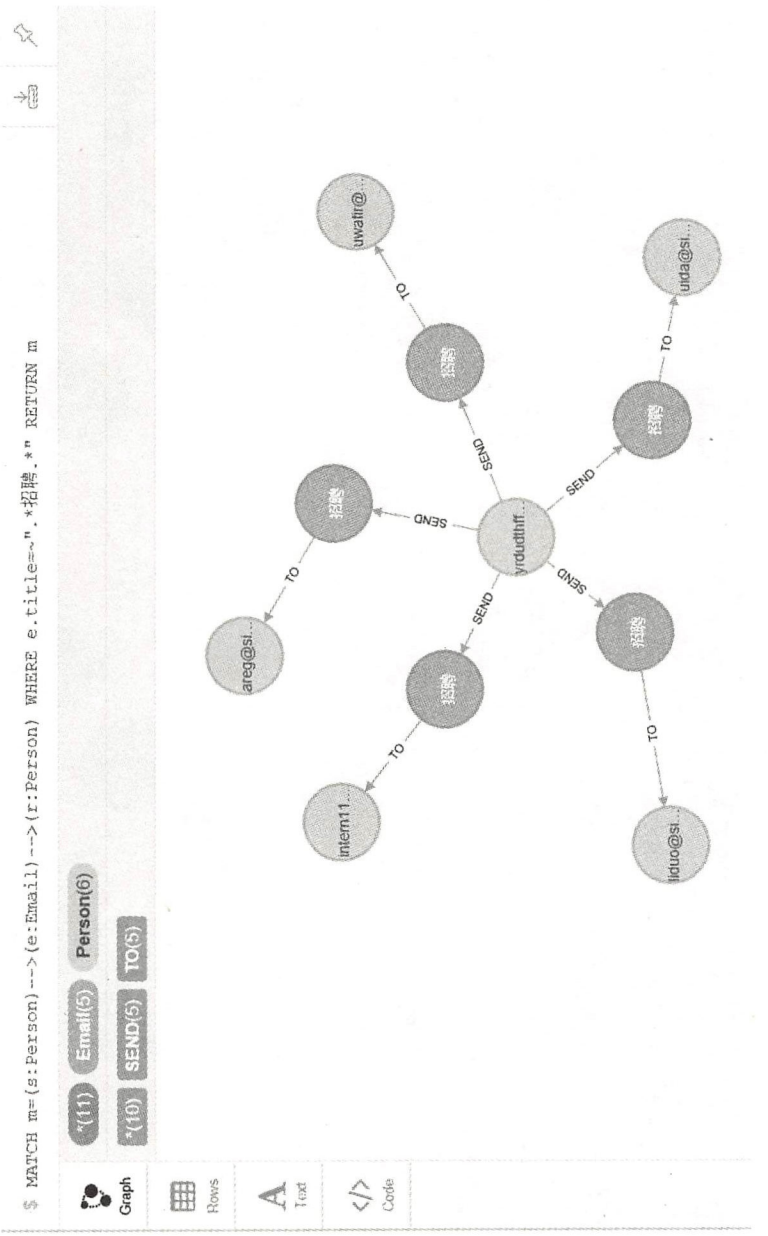


图 7-23 “招聘”邮件图谱

7.5 工商企业图谱

本节重点介绍 Neo4j 在工商企业信息管理中的应用。

自从李克强总理在 2015 年 3 月政府工作报告中提出“大众创业、万众创新”及 2015 年 6 月 16 日《国务院关于大力推进大众创业万众创新若干政策措施的意见》发布以来，我国商事制度环境获得了持续改进和优化，激发了大家的创业创新热潮。截至 2016 年 12 月 31 日，我国各类存量市场主体 8,705.4 万户，2016 年新设各类市场主体 1,651.3 万户，同比增幅达 11.6%，2016 年新增企业 552.8 万户，日均新增 1.51 万户。2016 年国家企业信用信息公示系统访问量和查询量分别达 227.93 亿人（次）和 114.19 亿人（次），日均访问量和查询量分别为 6,244.6 万人（次）和 3,128.4 万人（次）。与此同时，根据全国工商和市场监管部门的统计数据，2016 年共有 515.1 万户市场主体被列入经营异常名单，各类经济违法违章案件达到 51.6 万件，案值 72.1 亿元，日均发生 1,413 件。2016 年，全国工商总局各地分局对高法提供的 336.4 名“老赖”依法进行法人、董事和监事等任职限制，目前已经成功限制“老赖”7.1 万人（次）。根据全国工商和市场监管部门的统计数据，2016 年共受理消费投诉 166.7 万件，日均投诉达 4,567 件，涉及争议金额 40.6 亿元，为消费者挽回经济损失 18.2 亿元[32]。从上面的数据可以看到，我国经济系统是一个复杂巨系统，传统关系型数据库在发现经济问题线索、风险预警等方面面临着一定的挑战，市场主体的工商信息图谱需要一种新的技术来表达。

在考虑到近几年来企业融资诈骗、P2P 互联网金融公司“跑路”、企业违法犯罪事件频发，如何结合工商企业信息和财务、税务、社保、互联网等信息，及时高效地识别出高风险的企业或法人、股东等，仍然是一件十分具有挑战性的工作。在本节仅考虑利用 Neo4j 展示工商企业信息，尽管信息维度不是最丰富的，但通过 Neo4j 却已经可以在同样的数据集中挖掘出更多的隐藏价值。

1. 工商数据节点及关系集

（1）上级公司节点：中国石油化工集团公司；法人：王某。

（2）本级公司节点：中国石油化工股份有限公司；法人：王某；董事：张某、樊某、李某、阎某、蒋某、章某、汤某、戴某、焦某；董事长：王某；监事会主席：刘某；监事：邹某、刘某云、周某、俞某、王某钧。

（3）下级子公司节点：北京国际信托有限公司、中国石化国际事业有限公司、中国石化天然气有限责任公司、福建炼油化工有限公司、中国石化润滑油有限公司、广东中源贸易有限公司等 99 家子公司。

（4）孙公司节点：北京国际信托有限公司的子公司有广州市天马河房地产开发有限公司、泛华工程有限公司、天津泰达科技投资股份有限公司、国都证券股份有限公司；中国石化天然气有限责任公司的子公司有江西省投资燃气有限公司、湖北省天然气发展有限公司、华恒能源有限公司。

（5）本级公司节点的董事——樊某：北京国讯经科咨询中心（樊某监事，王某扬总经理、参股，乔某法人、执行董事、参股）、北京五十人论坛顾问有限公司（樊某参股，朱某监事，徐某执行董事、参股、总经理、法人）。

[32] http://www.gov.cn/xinwen/2017-01/19/content_5161163.htm#1

2. 工商数据节点及关系的 Neo4j 表达

```
create (中化集团:公司{名字:"中化集团"}),
 (中国石化:公司{名字:"中国石化"}),
 (北京信托:公司{名字:"北京信托"}),
 (中化国际:公司{名字:"中化国际"}),
 (中化天然气:公司{名字:"中化天然气"}),
 (福建炼化:公司{名字:"福建炼化"}),
 (中化润滑:公司{名字:"中化润滑"}),
 (广东中贸:公司{名字:"广东中贸"}),
 (广州天马:公司{名字:"广州天马"}),
 (泛华公司:公司{名字:"泛华公司"}),
 (泰达科技:公司{名字:"泰达科技"}),
 (国都证券:公司{名字:"国都证券"}),
 (江西投燃:公司{名字:"江西投燃"}),
 (湖北天然气:公司{名字:"湖北天然气"}),
 (华恒能源:公司{名字:"华恒能源"}),
 (中化集团)-[:控股]->(中国石化),
 (中国石化)-[:控股]->(北京信托),
 (中国石化)-[:控股]->(中化国际),
 (中国石化)-[:控股]->(中化天然气),
 (中国石化)-[:控股]->(福建炼化),
 (中国石化)-[:控股]->(中化润滑),
 (中国石化)-[:控股]->(广东中贸),
 (北京信托)-[:控股]->(广州天马),
 (北京信托)-[:控股]->(泛华公司),
 (北京信托)-[:控股]->(泰达科技),
 (北京信托)-[:控股]->(国都证券),
 (中化天然气)-[:控股]->(江西投燃),
 (中化天然气)-[:控股]->(湖北天然气),
 (中化天然气)-[:控股]->(华恒能源),
 (王某:高管{名字:"王某"}),
 (张某:高管{名字:"张某"}),
 (樊某:高管{名字:"樊某"}),
 (李某:高管{名字:"李某"}),
 (阎某: 高管{名字:"阎某"}),
 (蒋某:高管{名字:"蒋某"}),
 (章某:高管{名字:"章某"}),
 (汤某:高管{名字:"汤某"}),
 (戴某:高管{名字:"戴某"}),
 (焦某:高管{名字:"焦某"}),
 (刘某:高管{名字:"刘某"}),
 (邹某:高管{名字:"邹某"}),
 (刘某云:高管{名字:"刘某云"}),
 (周某:高管{名字:"周某"}),
 (俞某:高管{名字:"俞某"}),
 (王某钧:高管{名字:"王某钧"}),
 (王某)-[:法人]->(中化集团),
 (王某)-[:法人]->(中国石化),
 (张某)-[:董事]->(中国石化),
 (樊某)-[:董事]->(中国石化),
```

```
(李某)-[:董事]->(中国石化),
(阎某)-[:董事]->(中国石化),
(蒋某)-[:董事]->(中国石化),
(章某)-[:董事]->(中国石化),
(汤某)-[:董事]->(中国石化),
(戴某)-[:董事]->(中国石化),
(焦某)-[:董事]->(中国石化),
(王某)-[:董事长]->(中国石化),
(刘某)-[:监事会主席]->(中国石化),
(邹某)-[:监事]->(中国石化),
(刘某云)-[:监事]->(中国石化),
(周某)-[:监事]->(中国石化),
(俞某)-[:监事]->(中国石化),
(王某钧)-[:监事]->(中国石化),
(北京国讯:公司{名字:"北京国讯"}),
(五十人论坛:公司{名字:"五十人论坛"}),
(王某扬:高管{名字:"王某扬"}),
(乔某:高管{名字:"乔某"}),
(朱某:高管{名字:"朱某"}),
(徐某:高管{名字:"徐某"}),
(樊某)-[:监事]-> (北京国讯),
(王某扬)-[:总经理]-> (北京国讯),
(王某扬)-[:参股]-> (北京国讯),
(乔某)-[:法人]-> (北京国讯),
(乔某)-[:执行董事]-> (北京国讯),
(乔某)-[:参股]-> (北京国讯),
(樊某)-[:参股]-> (五十人论坛),
(朱某)-[:监事]-> (五十人论坛),
(徐某)-[:执行董事]-> (五十人论坛),
(徐某)-[:总经理]-> (五十人论坛),
(徐某)-[:参股]-> (五十人论坛),
(徐某)-[:法人]-> (五十人论坛)
```

3. 展示整个工商信息图谱

（1）执行程序。

```
MATCH p=()-[]-(n:`公司`)-[]-()-[]-() RETURN p
```

（2）执行结果（如图 7-24 所示）。

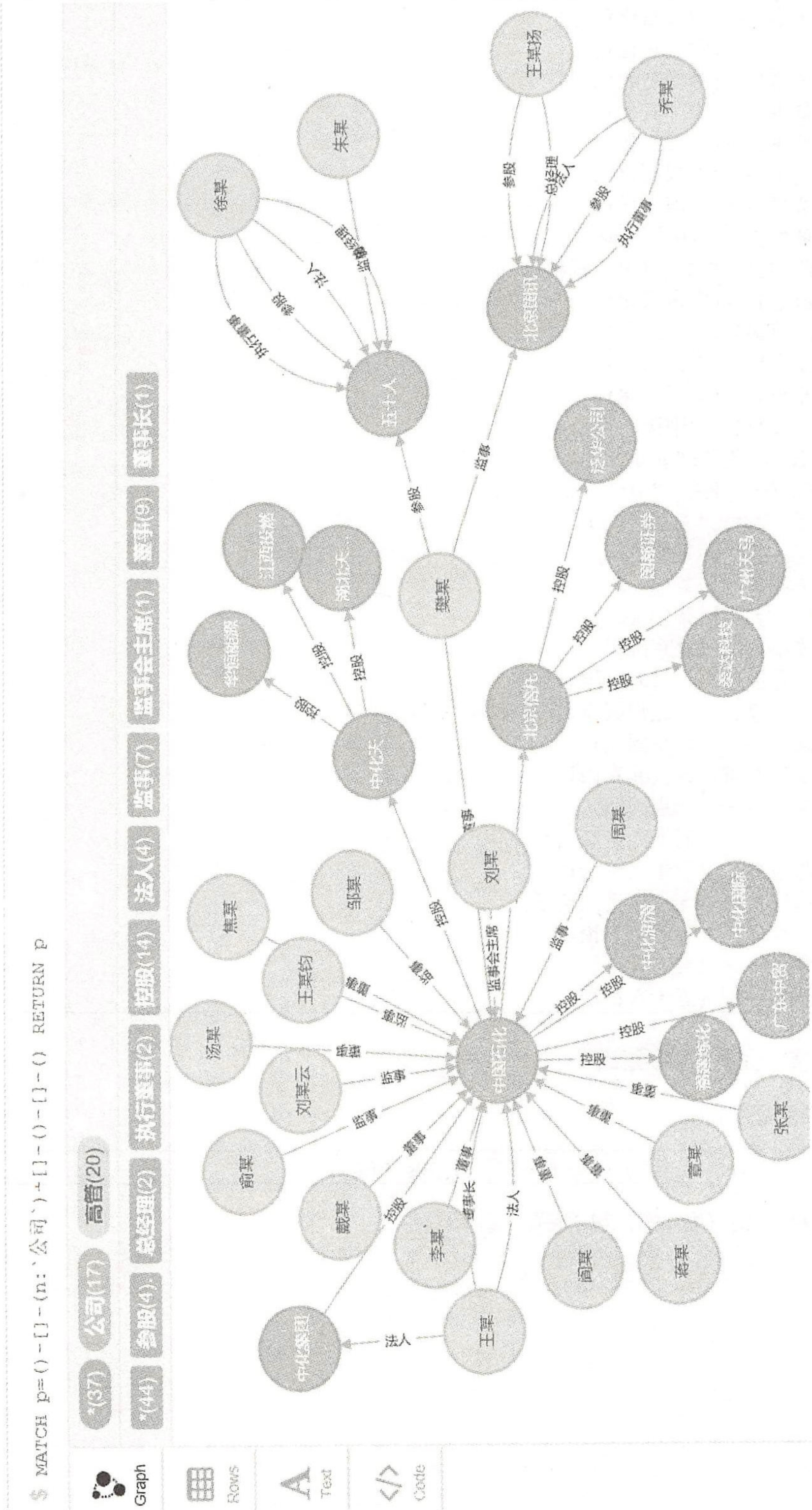


图 7-24　展示公司的工商图谱

4. 抽取关联公司图谱

（1）功能：用 Neo4j 展示中国石化的母公司、含有孙公司的子公司和孙公司的关系图谱。

（2）执行程序。

```
MATCH p=()-[:`控股`]->(:公司{名字:"中国石化"}) WITH p
MATCH q= (:公司{名字:"中国石化"})-[:`控股`]->()-[:`控股`]->()
RETURN p,q
```

（3）执行结果（如图 7-25 所示）。

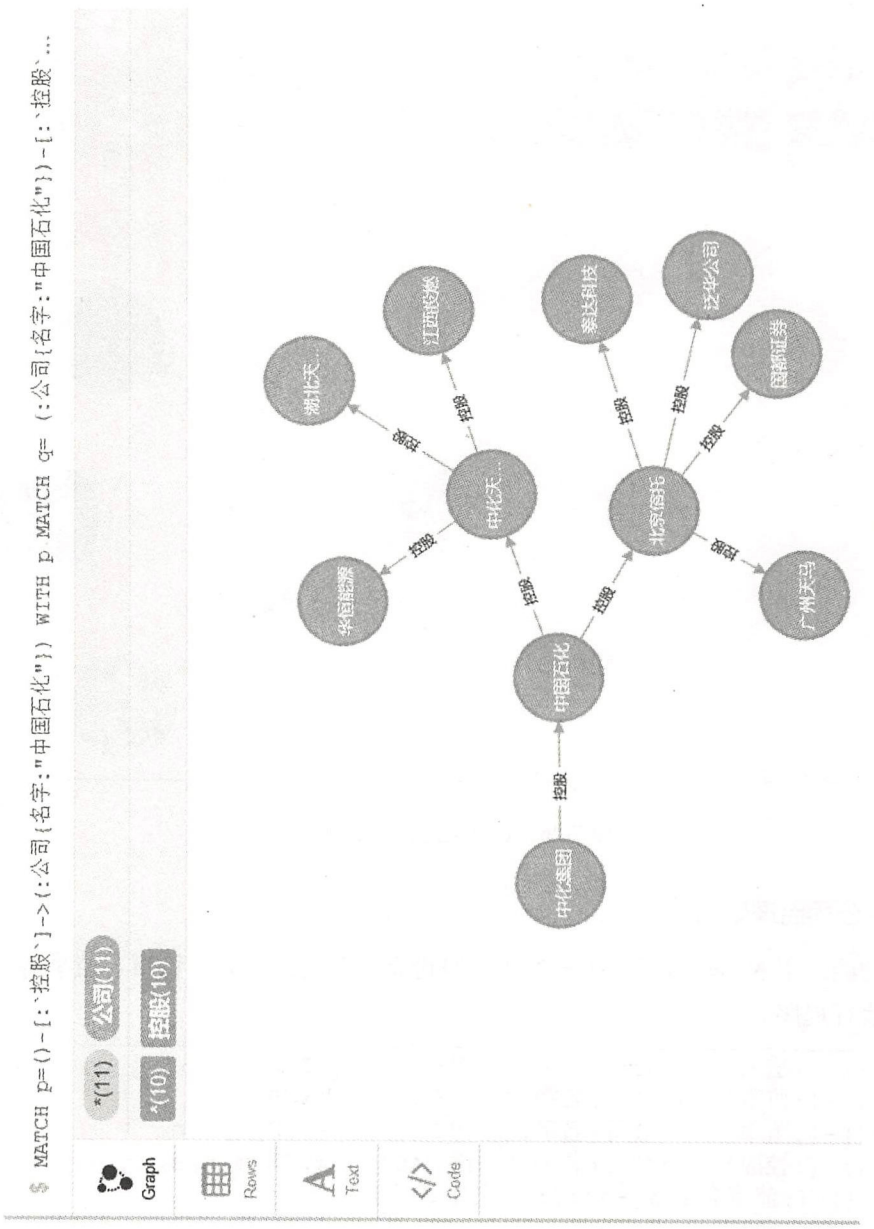


图 7-25　展示母子孙公司图谱

5. 展示高管个人信息

（1）功能：用 Neo4j 展示中国石油化工股份有限公司的董事樊某的所有关系图谱。

（2）执行程序。

```
MATCH (n:高管{名字:"樊某"})-[]->(s) RETURN n,s
```

（3）执行结果（如图 7-26 所示）。

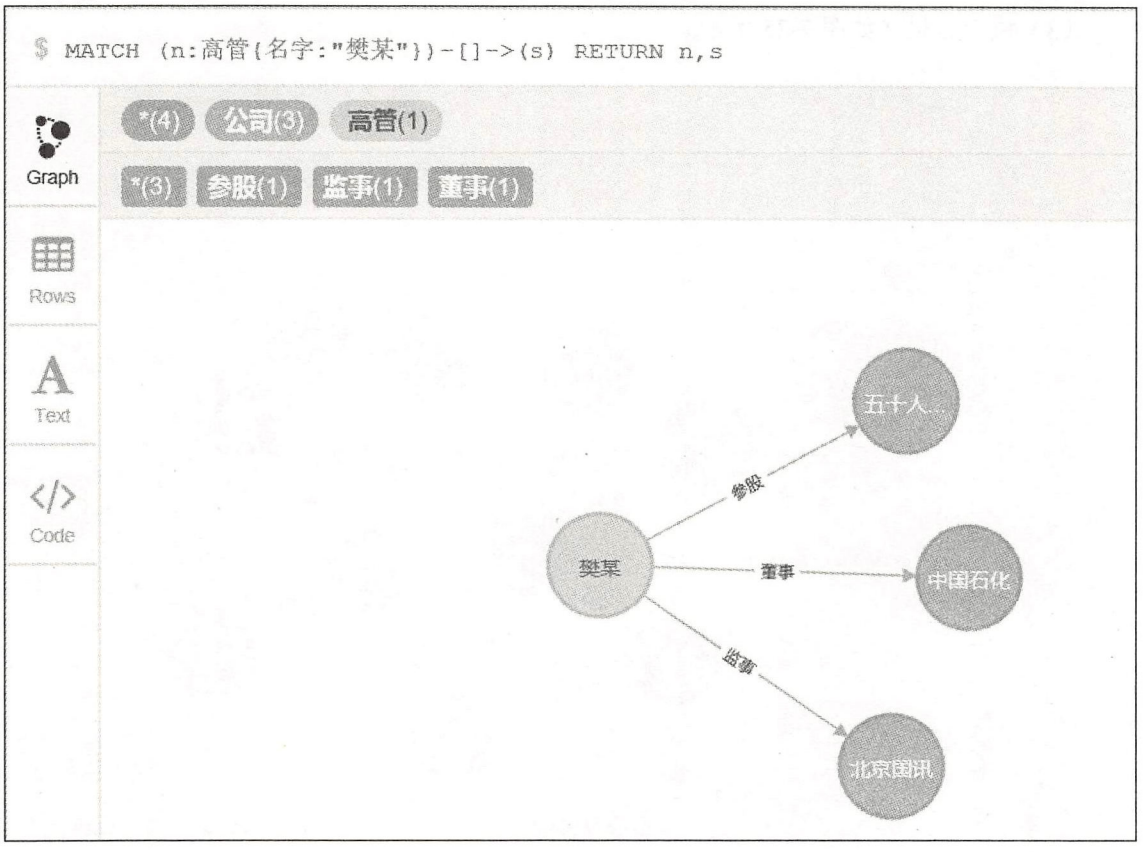


图 7-26　查询某高管的工商信息

6. 展示公司治理架构

（1）功能：用 Neo4j 展示中国石油化工股份有限公司的法人、董事、监事等治理架构。

（2）执行程序。

```
MATCH n=()-[:法人]->(:公司{名字:"中国石化"}) WITH n
MATCH m=()-[:监事]->(:公司{名字:"中国石化"}) WITH m,n
MATCH s=()-[:董事]->(:公司{名字:"中国石化"}) WITH s,m,n
MATCH r=()-[:控股]->(:公司{名字:"中国石化"}) WITH m,n,s,r
MATCH  t=()-[:监事会主席]->(:公司{名字:"中国石化"})
RETURN  m,n,s,r,t
```

（3）执行结果（如图 7-27 所示）。

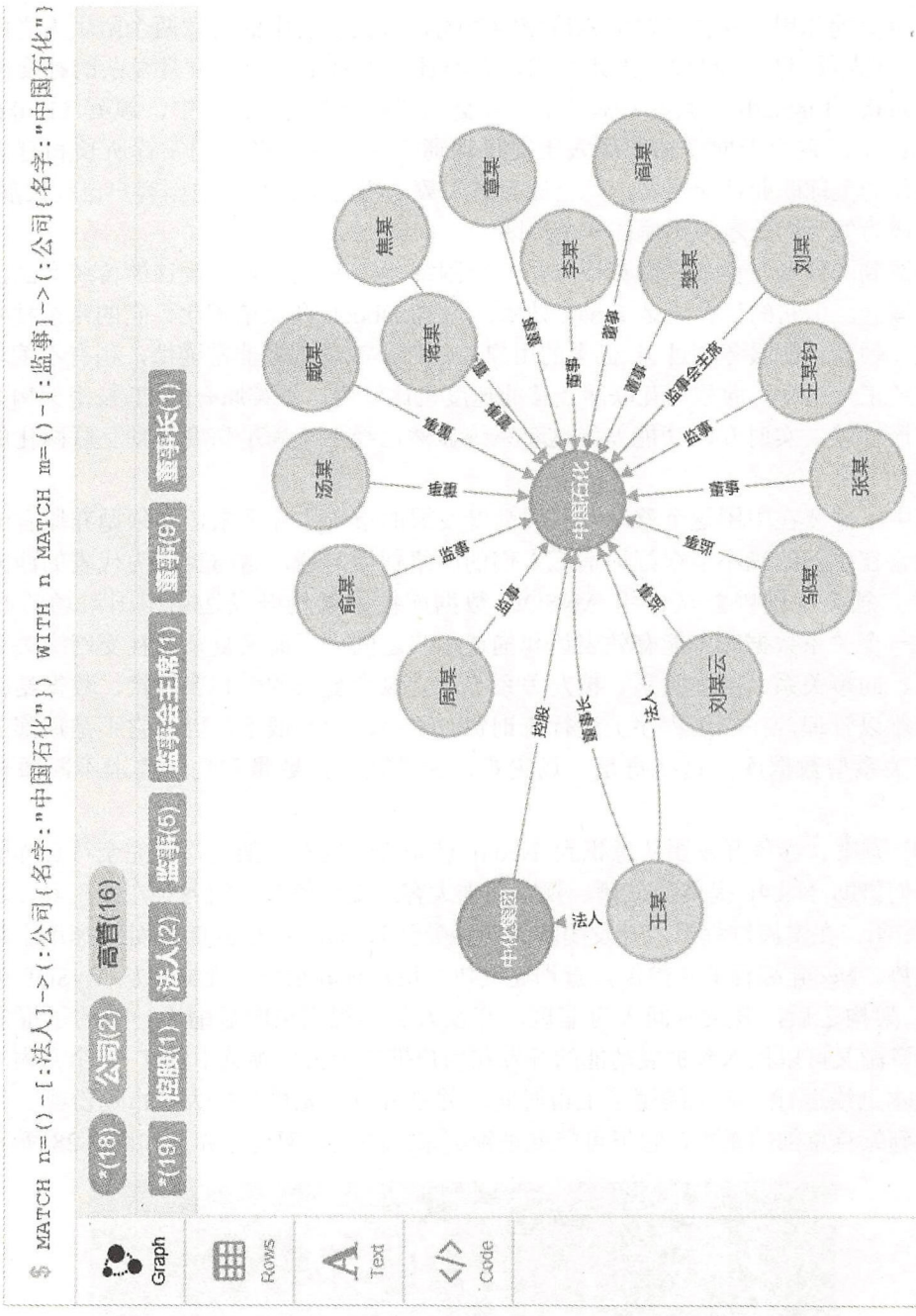


图 7-27　展示公司的治理结构

7.6 社交网络

在全球化的今天，竞争与合作是人类的基本生存方式，社交在人类生产、生活和学习中

起着十分重要的作用。根据著名的六度空间理论，人类社会中的任意两个陌生人之间最多通过 6 个中间人就可以连起来。由此可见，人类社会本身是一个巨大且复杂的社交网络。腾讯、Facebook、LinkedIn（领英）都是全球社交领域的巨头公司。其中，领英（LinkedIn）创建于 2002 年，致力于向全球职场人士提供沟通平台，并协助他们发挥所长和有效开展工作。领英作为全球职业社交网站，其全球会员人数已超过 4.5 亿，中国客户已经发展到 2000 多万人，成为领英增长最为迅速的一个地区。

领英遇到的问题与挑战。领英原本是一个围绕 PC 和电子邮箱设计的传统产品，尽管它有社交的概念，但它的后台还是 Email 技术。与 Facebook 和微信相比，它的系统技术架构太过于传统，领英目前服务超过 4 亿多的用户，任何小改动都要非常谨慎，稍有不慎就可能影响其他系统正常运行，向移动互联网迁移的速度也比较慢。领英原有的 IT 技术架构开发周期长，可扩展性差，实时互动功能差，这套系统显然已经不能满足高度移动互联网化的中国市场的需要。

领英中国想要在中国这个移动互联网高度发展的市场生存下来，必须抛弃原有的技术架构，因为它在中国的确不能很好地适应人们的网络使用习惯，这与美国为代表的西方国家有很大差异。领英中国曾尝试使用 MySQL 数据库技术架构开发适用中国市场的产品“赤兔”，但一个关系数据库不能有效表达和描述用户之间广泛而又复杂的社交网络关系，比如同学关系、同事关系、同乡关系、校友关系等。它包含复杂的表和表连接，数据结构显得非常复杂且难以管理，在应用程序上进行查询费时费力，效率低下，因为它不是局部而是全局搜索整个关系型数据库，这简直是一场灾难，尤其在用户数量和功能需求不断增长的情况下。

Neo4j 方案。赤兔开发团队意识到 Neo4j 就是赤兔的技术架构，它完全符合赤兔的产品需求。他们借助 Neo4j 提供的书籍、视频和强大客户支持等在内的各种在线工具，迅速了解和掌握 Neo4j。在实施过程中，开发团队深切感受到 Neo4j 与 MySQL 在处理速度上相比具备的巨大优势。Neo4j 编程非常简单，查询速度快，用户体验很好。在赤兔从 MySQL 架构切换到 Neo4j 架构之后，开发时间大为缩短，开发人员的规模也明显缩小，节约了很多资金成本，这些资源又可以投入到扩展功能的开发和用户推广上去。领英中国在 4 个月内就启动了第一个版本上线运行，大大缩短了上市时间，并抢占市场先机和扩大了市场份额。目前赤兔越来越受到领英总部的重视，它很可能就是领英未来的技术架构，赤兔如图 7-28 所示。



图 7-28　赤兔

1. 案例 1——赤兔好友列表中的关系是 1 度关系

在手机 APP 商店下载“赤兔”APP，安装，注册，登录。赤兔 1 度好友列表如图 7-29 所示。

2. 人脉列表中的人际关系网络展示到 3 度，如图 7-30 所示



图 7-29　赤兔好友列表 1 度关系

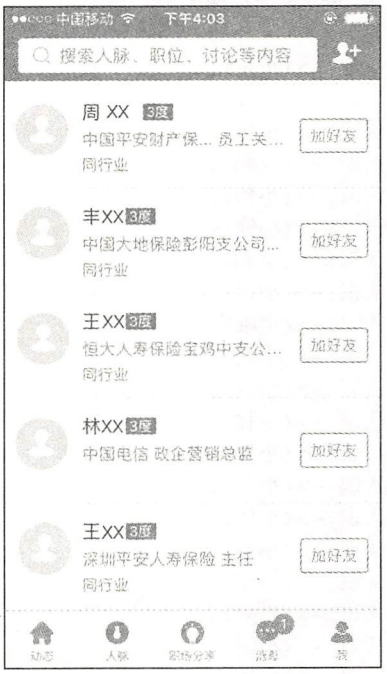


图 7-30　3 度人脉关系列表

3. 应用 Neo4j 创建示范案例节点及关系

（1）数据节点和关系集

个人节点：小北、小菲、小鹏、小颖、小兰、小锋、小讯、小东、小唯、小窦、小齐、小林、小锐、小伟、小玲。

节点之间的关系：小讯认识小窦、小齐、小林、小鹏、小伟、小锋；小菲认识小鹏、小锋、小唯；小锋认识小北、小兰；小东认识小菲、小锐、小林；小鹏认识小颖；小北认识小兰；小颖认识小东；小唯认识小鹏、小锐；小伟认识小玲。

（2）数据节点和关系集的 Neo4j 表示

```
CREATE
(小北:朋友圈{姓名: "小北"}),
(小菲:朋友圈{姓名: "小菲"}),
(小鹏:朋友圈{姓名: "小鹏"}),
(小颖:朋友圈{姓名: "小颖"}),
(小兰:朋友圈{姓名: "小兰"}),
(小锋:朋友圈{姓名: "小锋"}),
```

```
(小讯:朋友圈{姓名: "小讯"}),
(小东:朋友圈{姓名: "小东"}),
(小唯:朋友圈{姓名: "小唯"}),
(小窦:朋友圈{姓名: "小窦"}),
(小齐:朋友圈{姓名: "小齐"}),
(小林:朋友圈{姓名: "小林"}),
(小锐:朋友圈{姓名: "小锐"}),
(小伟:朋友圈{姓名: "小伟"}),
(小玲:朋友圈{姓名: "小玲"}),
(小讯)-[:认识]->(小窦),
(小讯)-[:认识]->(小齐),
(小讯)-[:认识]->(小林),
(小讯)-[:认识]->(小鹏),
(小讯)-[:认识]->(小伟),
(小讯)-[:认识]->(小锋),
(小菲)-[:认识]->(小鹏),
(小菲)-[:认识]->(小锋),
(小菲)-[:认识]->(小唯),
(小锋)-[:认识]->(小北),
(小锋)-[:认识]->(小兰),
(小东)-[:认识]->(小林),
(小东)-[:认识]->(小锐),
(小东)-[:认识]->(小菲),
(小鹏)-[:认识]->(小颖),
(小北)-[:认识]->(小兰),
(小颖)-[:认识]->(小东),
(小唯)-[:认识]->(小鹏),
(小唯)-[:认识]->(小锐),
(小伟)-[:认识]->(小玲),
```

4. 展示朋友圈

（1）功能：展示整个朋友圈关系图。

（2）执行程序。

```
MATCH n=(:朋友圈{姓名:"小锋"})-[*..6]-() RETURN n
```

（3）执行结果（如图 7-31 所示）。

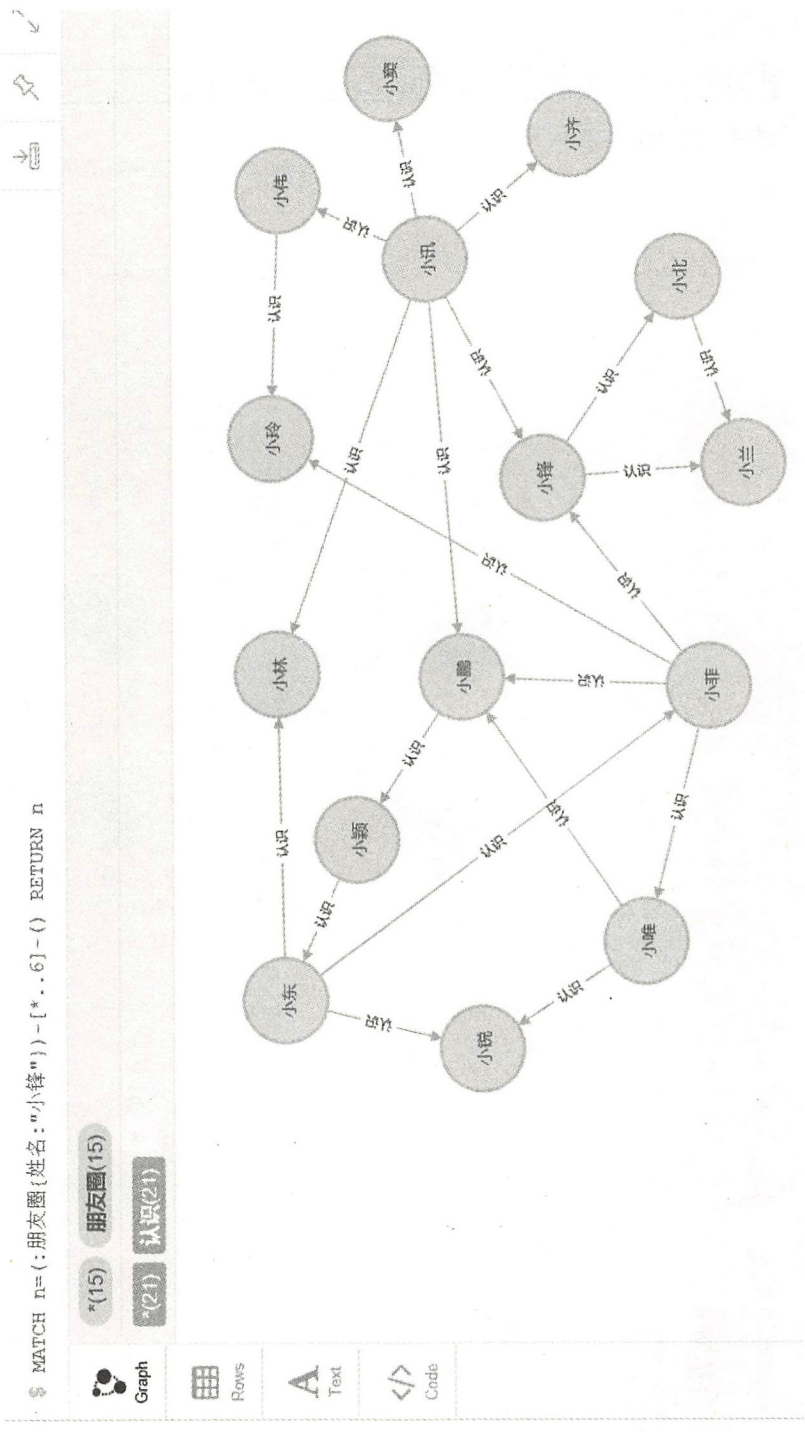


图 7-31　展示小锋的朋友圈

5. 展示一度关系

（1）功能：展示小讯直接认识的朋友。

（2）执行程序。

```
MATCH n=(:朋友圈{姓名:"小讯"})-[:认识]->() RETURN n
```

（3）执行结果（如图 7-32 所示）。

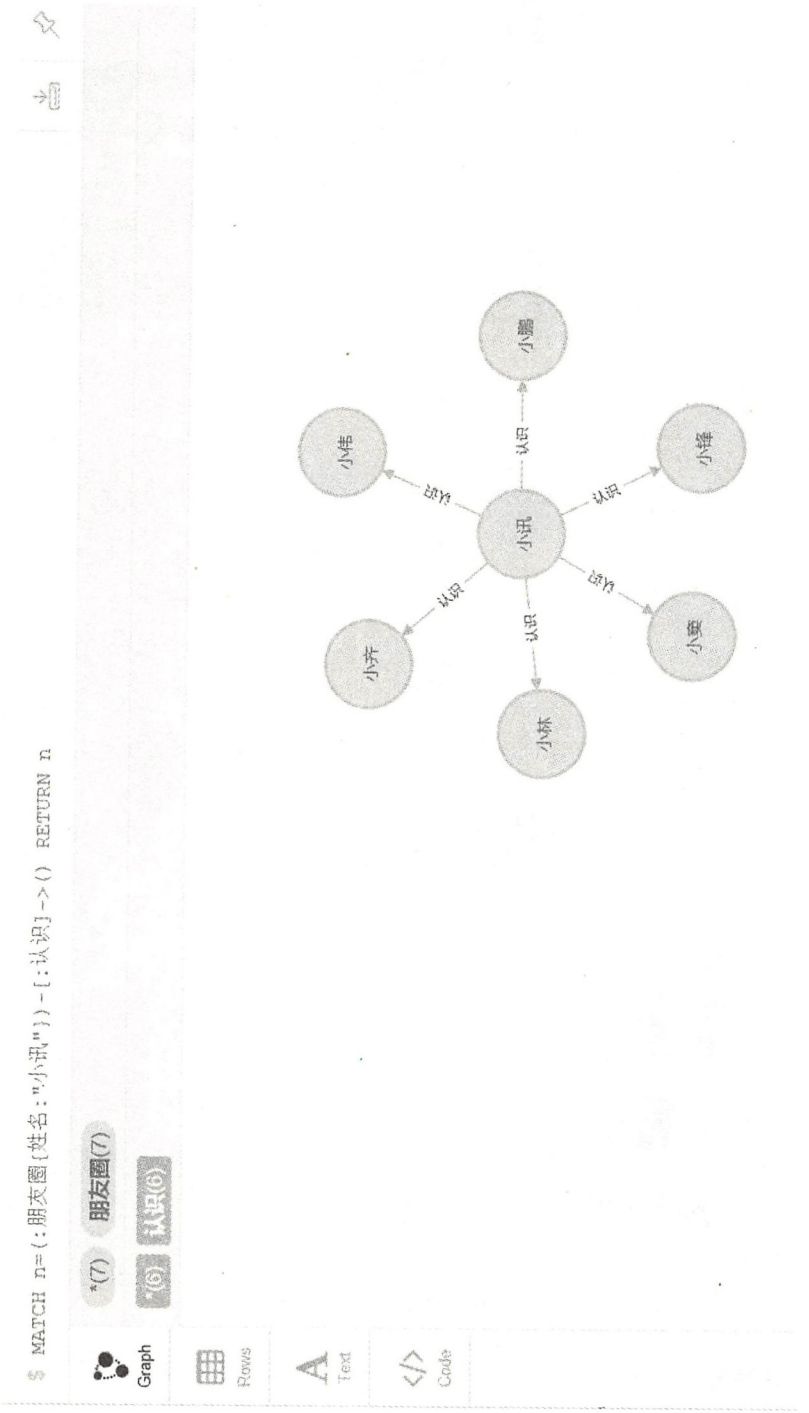


图 7-32　展示小讯的 1 度朋友关系图谱

6. 展示二度关系

（1）功能：展示小讯认识的朋友以及小讯认识的朋友的朋友。

（2）执行程序。

```
MATCH n=(小讯:朋友圈{姓名:"小讯"})-[*..2]->() RETURN n
```

（3）执行结果（如图 7-33 所示）。

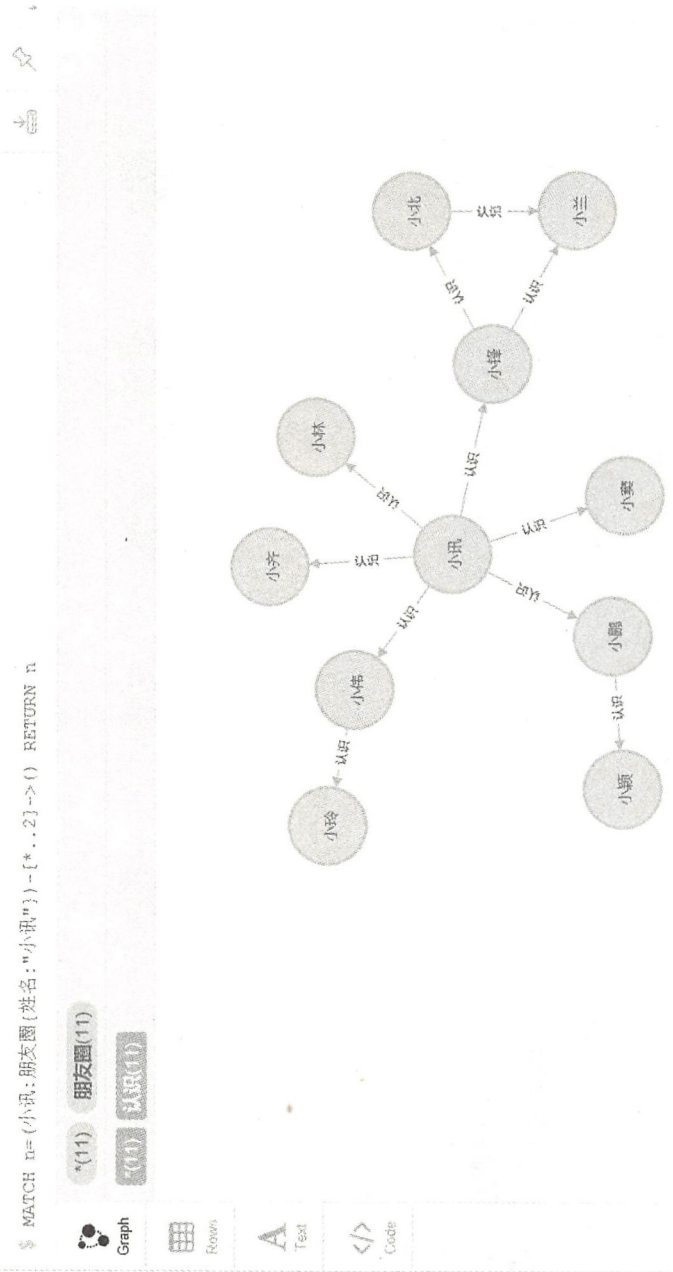


图 7-33　展示小讯的 2 度朋友关系图谱

7. 依托社交网络搜寻两人之间的最短熟人路径。

（1）功能：寻找小讯与小锐之间的最短路径。

（2）执行程序。

```
MATCH n= shortestPath((小讯:朋友圈{姓名:"小讯"})-[*..6]->(小锐:朋友圈{姓名:"小锐"}))
RETURN n
```

（3）执行结果（如图 7-34 所示）。

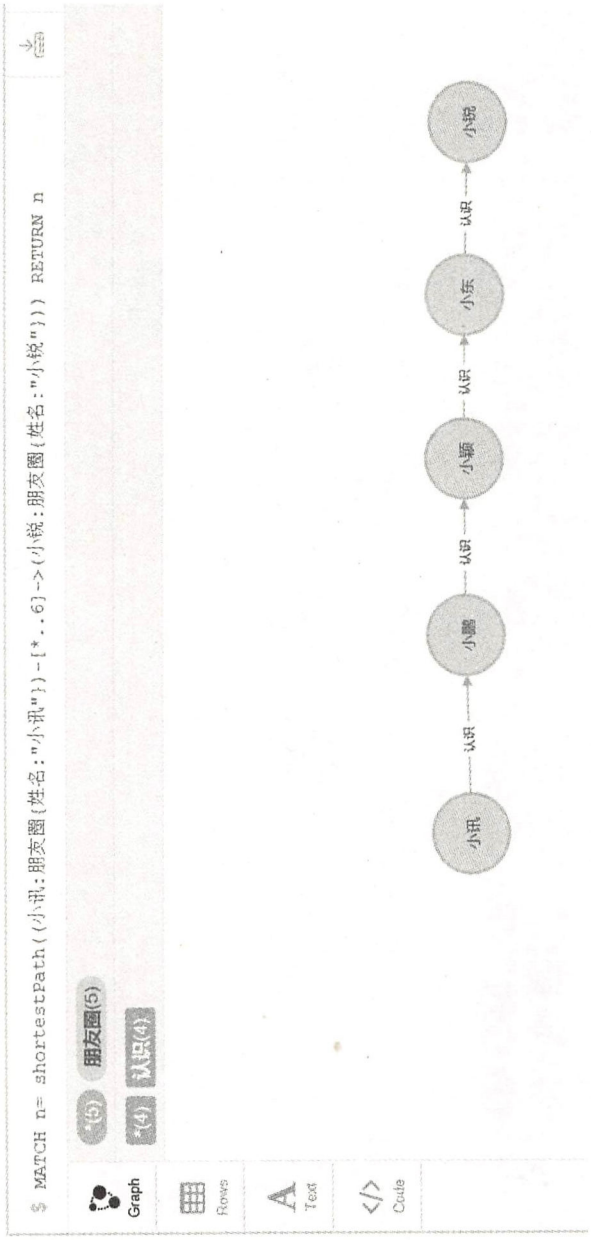


图 7-34　寻找两人之间的最短路径

8. 搜索两个人之间所有的最短路径

（1）功能：寻找小讯与小菲之间所有的最短路径。

（2）执行程序。

```
MATCH n= allshortestPaths((小讯:朋友圈{姓名:"小讯"})-[*..6]-(小菲:朋友圈
{姓名:"小菲"}))
RETURN n
```

（3）执行结果（如图 7-35 所示）。

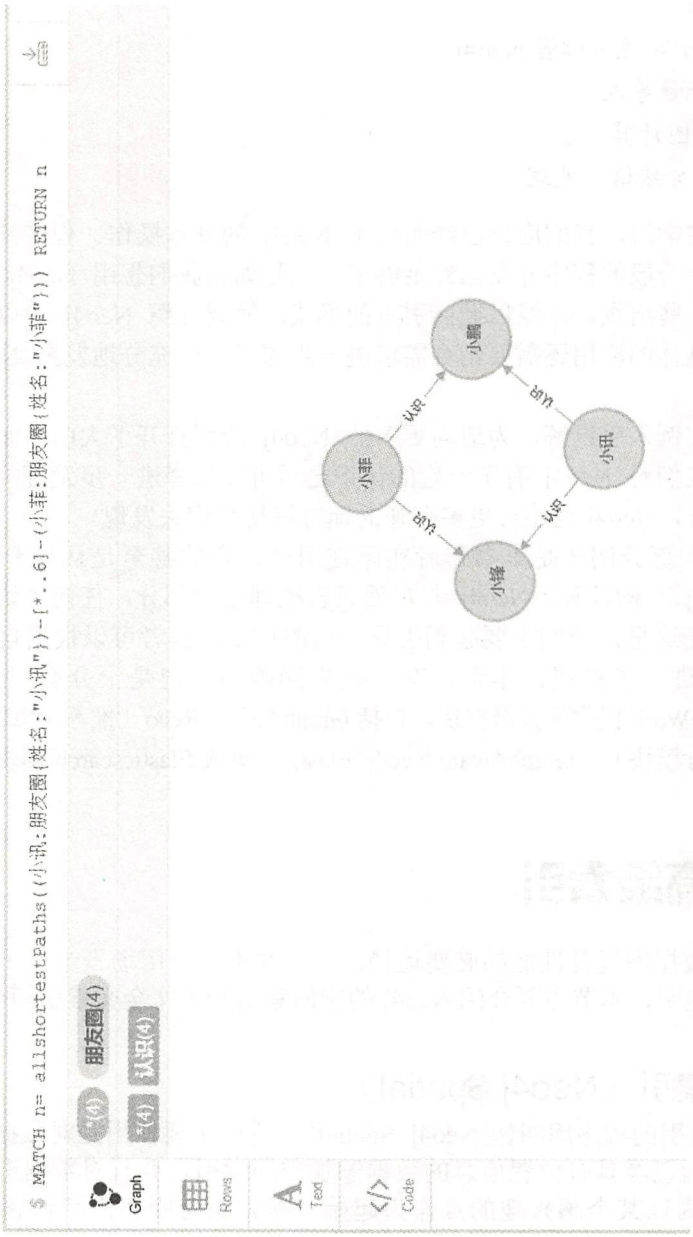


图 7-35　寻找两人之间所有的最短路径

第 8 章

◂Neo4j 高级应用▸

本章包括以下内容：

- 高级索引
- 在 Docker 环境下部署 Neo4j
- 自定义批量导入
- Neo4j 与图计算
- Neo4j 与自然语言处理

通过前几章的学习，我们应该已经明白了 Neo4j 的基本操作、程序开发、数据库管理等常用功能。这对于一般的程序开发已经足够了，但是如果我们想用 Neo4j 探索更多的应用，可以在本章得到一些启发。本章以抛砖引玉的形式，简要介绍 Neo4j 可以应用到的一些具有代表性的领域，具体的应用还需要结合需求进一步探索，以充分地发挥 Neo4j 的更深层次、更高级别的功能。

本章以几个实例入手讲解，为通向更多的 Neo4j 应用打开了大门。而且本章包含大量的实践知识，是在我们对 Neo4j 有了一定的认识之后可以选择进一步发掘的部分。当然本章也有不够周全的地方，Neo4j 也还有更多有趣的地方等待我们去发现。

本章中的绝大部分例子是本书作者实际使用过，有的甚至是从作者的应用案例中摘录的。特别是 Neo4j 与图计算、Neo4j 与自然语言处理这两部分，任何一块都值得用大量的篇幅来详细描述。在这里，我们主要起到引导入门的作用，读者可以根据自己知识的实际掌握程度和自身需求进一步扩展。本章还有一些未涵盖的、但是十分有用的高级应用，例如 GraphAware FrameWork 提供的扩展模块，包括 GraphAware Reco（推荐引擎模块）、GraphAware Runtime（实时运行模块）、GraphAware Neo4j2Elastic（集成 Elasticsearch 模块）等。

8.1 高级索引

索引是每个数据库提升性能的必要途径之一，在现实应用场景中，不同的业务查询需要建立不同类别的索引，本节主要介绍 Neo4j 的空间索引和中文全文索引两个部分。

8.1.1 空间索引（Neo4j Spatial）

Neo4j 空间索引的程序库叫做 Neo4j Spatial[33]，它的主要作用是对数据进行空间操作。程序开发人员可以向已经具有位置信息的数据添加空间索引，并且对数据进行空间计算操作，例如在指定区域内以某个感兴趣的点作为起始，搜索指定距离内其他感兴趣的点。此外，

[33] https://github.com/neo4j-contrib/spatial

Neo4j Spatial 还提供了将数据导入 GeoTools[34]（Java 语言编写的开源 GIS 工具包），从而启动 GeoTools 的应用程序，比如 GeoServer[35]（共享空间地理数据的开源服务器）和 uDig[36]（开源桌面地理数据访问、编辑、呈现框架）。空间数据的存储结构如图 8-1 所示。

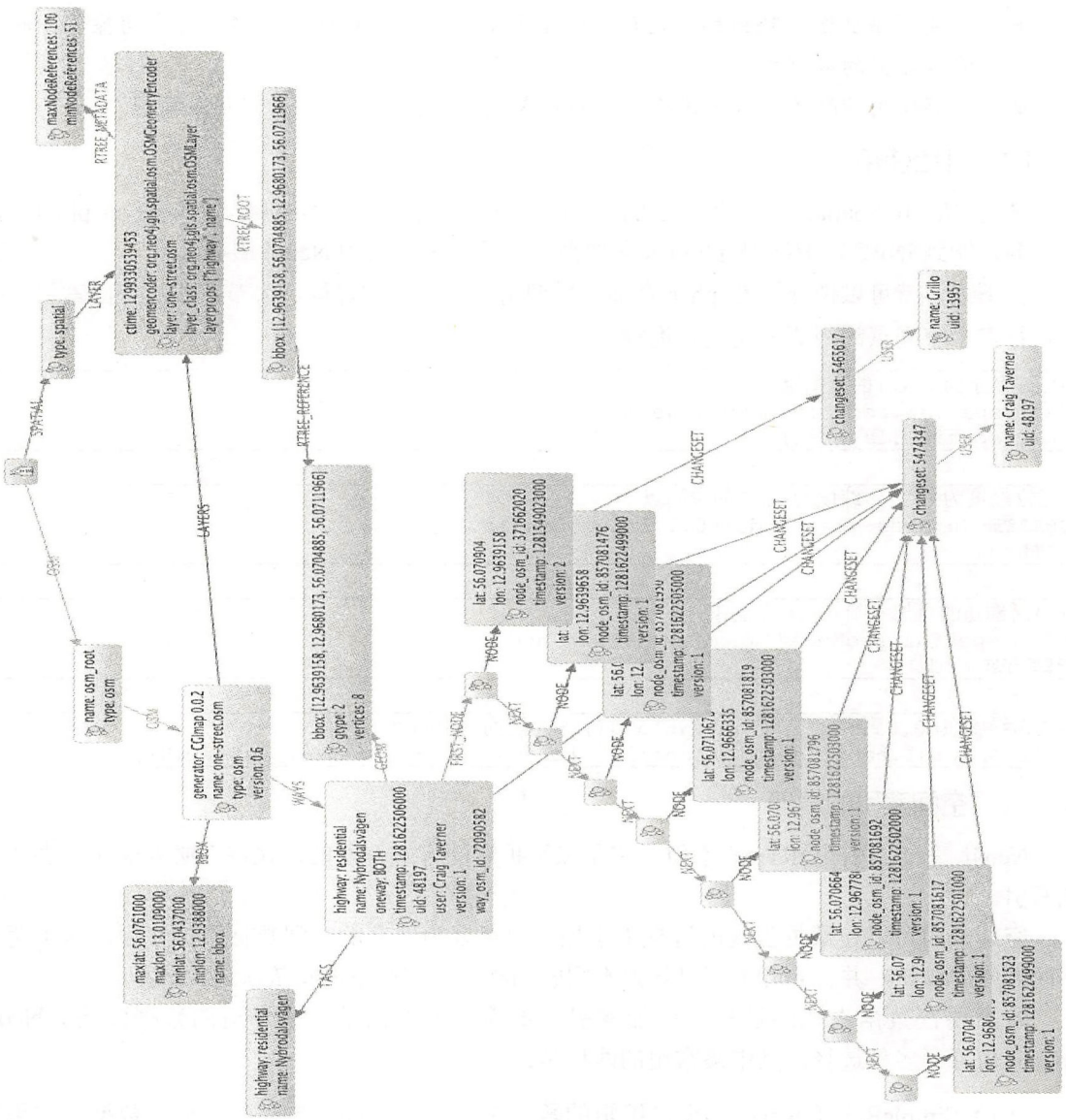


图 8-1　空间数据的存储结构

Neo4j Spatial 的主要功能包括：

- 可以将 ESRI Shapefile（SHP）和 Open Street Map（OSM）文件导入 Neo4j 的实用程序。

[34] http://geotools.org/
[35] http://geoserver.org/
[36] http://udig.refractions.net/

- 支持所有常见的几何类型：点、线、多边形等。
- 用于快速搜索几何形状的 RTree 结构。
- 支持搜索期间的拓扑操作（包含、属于、相交、覆盖等）。
- 只要提供从图形映射到几何形状的适配器，就可以对任何图形进行空间操作，而不管其数据的存储方式。
- 能够使用预配置的过滤器将单个图层或数据集拆分成多个子图层或视图。

8.1.1.1 开始使用

使用 Neo4j Spatial 的最简单的方法是获取 neo4j-spatial-*.**-neo4j-*.*.*-server-plugin[37]，并将其复制到$NEO4J_HOME/plugins 文件夹中，然后重新启动 Neo4j 服务。

接下来，就可以像使用 Cypher 查询一样调用 Neo4j 空间过程，为节点添加空间索引，并且执行多个空间点的距离、交叉查询等。

```
#创建名字为 geom 的点图层
CALL spatial.addPointLayer('geom')
CALL spatial.layers()
```

```
#建立经度为15.2、纬度为60.1的空间点
CREATE (n:Node {latitude:60.1,longitude:15.2})
WITH n
```

```
#将新创建的点加入到 geom 图层中
CALL spatial.addNode('geom',n) YIELD node
RETURN node
```

```
#查询经度在60.0到60.1之间，纬度在15.0到15.3之间的空间点
CALL spatial.bbox('geom',{lon:15.0,lat:60.0}, {lon:15.3, lat:61.0})
```

8.1.1.2 空间索引的原理

Neo4j 空间索引是 RTree 索引，它是以可扩展的方式开发的，允许在必要的时候添加其他索引。

空间索引可以在数据生成的过程中添加，也可以为现有的空间数据添加索引。这是两个不同的应用场景，并且实际上可以导致不同的图结构，因此下面将依次进行解释。

若需要在数据生成的过程中添加索引，最简单的方法是创建合适的数据图层。Neo4j Spatial 内置了多种选择，其中最常用的两种是：

（1）SimplePointLayer：一个可编辑的数据图层，仅允许向数据库添加点数据。如果只有点数据并且感兴趣的是邻近搜索，这是一个不错的选择。这一图层包括专门针对这种数据的实用程序方法。

（2）EditableLayer（Impl）：默认的可编辑图层实现，可以处理任何简单的几何类型。由于它是一个通用实现，并且无法事先知道数据模型的拓扑结构，因此将每个几何图形分别存储在单个节点的单个属性中。存储格式是 WKB（Well Known Binary），它是专门用于几何地理的二进制格式。

[37] https://github.com/neo4j-contrib/spatial/releases

8.1.1.3　图层和几何编码器

定义几何图形的集合叫做图层，其中包含可用于查询的索引。如果可以在图层中添加和修改图形，那么这个图层是可编辑图层（Editable Layer）。

DefaultLayer 是标准图层，利用 WKBGeometryEncoder 将所有未知几何类型存储为单个节点的一个二进制格式属性。

OSMLayer 是将开放街道地图（Open Street Map，OSM）用 OSM 模型存储为单个完全连接图的特殊层。该图层支持的几何图形包括点、线和多边形，因此无法导出为 Shapefile 格式，因为该格式每个图层为一种类型的几何图形。OSMLayer 扩展了 DynamicLayer，它允许使用任何数量的子层，每个子层都可以有特定的几何类型，并且是基于 OSM 标签过滤的。

8.1.1.4　空间过程

空间过程如表 8-1 所示。

表 8-1　Neo4j 空间过程

过程名	过程声明
spatial.addLayer	spatial.addLayer(name::STRING,type::STRING,encoderConfig::STRING)::(node::NODE)
spatial.addLayerWithEncoder	spatial.addLayerWithEncoder(name::STRING,encoder::STRING,encoderConfig::STRING)::(node::NODE)
spatial.addNode	spatial.addNode(layerName::STRING,node::NODE)::(node::NODE)
spatial.addNodes	spatial.addNodes(layerName::STRING,nodes::LIST.OF.NODE)::(node::NODE)
spatial.addPointLayer	spatial.addPointLayer(name::STRING)::(node::NODE)
spatial.addPointLayerWithConfig	spatial.addPointLayerWithConfig(name::STRING,encoderConfig::STRING)::(node::NODE)
spatial.addPointLayerXY	spatial.addPointLayerXY(name::STRING,xProperty::STRING,yProperty::STRING)::(node::NODE)
spatial.addWKT	spatial.addWKT(layerName::STRING,geometry::STRING)::(node::NODE)
spatial.addWKTLayer	spatial.addWKTLayer(name::STRING,nodePropertyName::STRING)::(node::NODE)
spatial.addWKTs	spatial.addWKTs(layerName::STRING,geometry::LIST.OF.STRING)::(node::NODE)
spatial.asExternalGeometry	spatial.asExternalGeometry(geometry::ANY)::(geometry::ANY)
spatial.asGeometry	spatial.asGeometry(geometry::ANY)::(geometry::ANY)
spatial.bbox	spatial.bbox(layerName::STRING,min::ANY,max::ANY)::(node::NODE)
spatial.closest	spatial.closest(layerName::STRING,coordinate::ANY,distanceInKm::FLOAT)::(node::NODE)
spatial.decodeGeometry	spatial.decodeGeometry(layerName::STRING,node::NODE)::(geometry::ANY)
spatial.getFeatureAttributes	spatial.getFeatureAttributes(name::STRING)::(name::STRING)
spatial.importShapefile	spatial.importShapefile(uri::STRING)::(node::NODE)
spatial.importShapefileToLayer	spatial.importShapefileToLayer(layerName::STRING,uri::STRING)::(node::NODE)
spatial.intersects	spatial.intersects(layerName::STRING,geometry::ANY)::(node::NODE)
spatial.layer	spatial.layer(name::STRING)::(node::NODE)
spatial.layerTypes	spatial.layerTypes()::(name::STRING,signature::STRING)
spatial.layers	spatial.layers()::(name::STRING,signature::STRING)
spatial.procedures	spatial.procedures()::(name::STRING,signature::STRING)
spatial.removeLayer	spatial.removeLayer(name::STRING)::VOID
spatial.setFeatureAttributes	spatial.setFeatureAttributes(name::STRING,attributeNames::LIST OF STRING)::(node::NODE)

（续表）

过程名	过程声明
spatial.updateFromWKT	spatial.updateFromWKT(layerName::STRING,geometry::STRING,geometryNodeId::INTEGER)::(node::NODE)
spatial.withinDistance	spatial.withinDistance(layerName::STRING,coordinate::ANY,distanceInKm::FLOAT)::(node::NODE,distance::FLOAT)

8.1.1.5 空间索引查询类型

Neo4j Spatial 索引是 RTree 索引，支持扩展开发，可以允许在必要的时候添加其他索引。RTree 索引可以实现的查询方式有：

- 包含（Contain）
- 覆盖（Cover）
- 被覆盖（Covered By）
- 交叉（Cross）
- 不相交（Disjoint）
- 相交（Intersect）
- 相交窗口（Intersect Window）
- 交叠（Overlap）
- 接触（Touch）
- 包含（Within）
- 在一定距离内（With Distance）

8.1.1.6 Java 构建 Neo4j 空间索引

Neo4j 自带一个用于导入 ESRI Shapefile 数据的实用程序。ShapefileImporter 将为每个导入的 Shapefile 创建一个新的图层，并将每个几何图形作为 WKB 存储在单个节点的单个属性中。每个图形的所有属性将作为该节点的其他属性存储，有关如何实现的更多信息，可以参考 WKBGeometryEncoder 类，但只是使用这个类就没必要了解它的实现细节。

1. 导入 shape 文件

下面代码将 roads.shp 导入到“layer_roads”图层中。该图层使用 RTree 实现索引，支持上述已有的空间查询操作。

```
//数据库文件位置
File storeDir = new File(dbPath);
GraphDatabaseService database = new
GraphDatabaseFactory().newEmbeddedDatabase(storeDir);
try (Transaction tx = database.beginTx()) {
    ShapefileImporter importer = new ShapefileImporter(database);
    //导入 shp 文件地址，生成图层名称
    importer.importFile("roads.shp", "layer roads");
    tx.success();
} finally {
    database.shutdown();
}
```

使用 Neo4j Spatial 过程调用，也可以达到同样的效果。其方法为：

```
CALL spatial.addWKTLayer('layer_roads','geometry')
CALL spatial.importShapefileToLayer('layer_roads','roads.shp')
```

结果如图 8-2 所示。

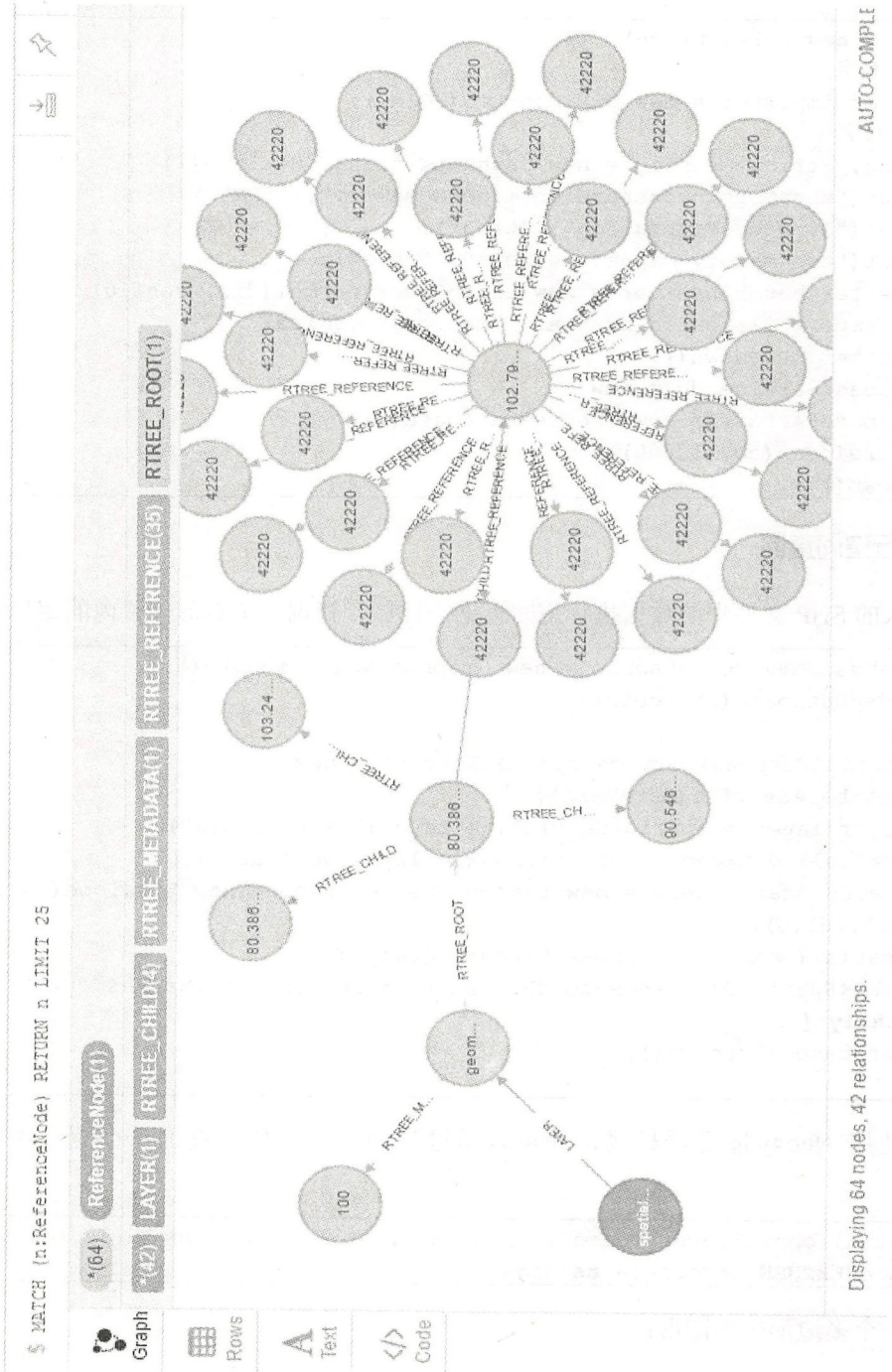


图 8-2　导入全国公路的 SHP 文件

2. 导入开放街道地图文件（OSM）

导入 OSM 文件要比导入 SHP 文件要复杂，因为导入 OSM 文件需要分两个过程执行，而第一个过程需要批处理导入。有关更详细的介绍可以参考 Neo4j Spatial 源码[38]中的 TestDynamicLayers 和 TestOSMImport 类中的单元测试，以获取导入 OSM 数据的最新代码。

```
File dir = new File(dbPath)
//设置图层名字
OSMImporter importer = new OSMImporter("OSM");
//批量导入配置
Map<String, String> config = new HashMap<String, String>();
config.put("neostore.nodestore.db.mapped_memory", "90M" );
config.put("dump_configuration", "true");
config.put("use_memory_mapped_buffers", "true");
BatchInserter batchInserter = new BatchInserterImpl(dir, config);
importer.importFile(batchInserter, "map.osm", false);
batchInserter.shutdown();
GraphDatabaseService db = new
GraphDatabaseFactory().newEmbeddedDatabase(dir);
importer.reIndex(db, 10000);
db.shutdown();
```

3. 执行空间查询

对导入的 SHP 文件中的数据执行空间查询，比如，查询一个矩形位置内的图形。

```
GraphDatabaseService database = new GraphDatabaseFactory().
newEmbeddedDatabase(storeDir);
    try {
        SpatialDatabaseService spatialService = new
SpatialDatabaseService(database);
        Layer layer = spatialService.getLayer("layer_roads");
        SpatialIndexReader spatialIndex = layer.getIndex();
        Search searchQuery = new SearchIntersectWindow(new Envelope(15.0,
15.3, 60.0, 61.0));
        spatialIndex.executeSearch(searchQuery);
        List<SpatialDatabaseRecord> results = searchQuery.getResults();
    } finally {
        database.shutdown();
    }
```

如果使用 Neo4j 服务器模式，还可以用过程调用。比如，查询一个矩形内的图形语句为：

```
CALL spatial.bbox('layer_roads', {lon:15.0,lat:60.0}, {lon:15.3, lat:61.0})
YIELD node RETURN node.name as name
```

查询一个多边形内的图形：

[38] https://github.com/neo4j-contrib/spatial

```
WITH "POLYGON((15.3 60.2, 15.3 60.4, 15.7 60.4, 15.7 60.2, 15.3 60.2))" as
polygon
CALL spatial.intersects('layer_roads',polygon) YIELD node
RETURN node.name as name
```

对导入的 OSM 文件数据进行空间查询：

```
GraphDatabaseService database = graphDb();
try {
    SpatialDatabaseService spatialService = new
SpatialDatabaseService(database);
    Layer layer = spatialService.getLayer("map.osm");
    LayerIndexReader spatialIndex = layer.getIndex();
    System.out.println("Have " + spatialIndex.count() + " geometries in " +
spatialIndex.getBoundingBox());
    Envelope bbox = new Envelope(12.94, 12.96, 56.04, 56.06);
    try(Transaction tx = database.beginTx()) {
        List<SpatialDatabaseRecord> results = GeoPipeline
            .startIntersectWindowSearch(layer, bbox)
            .toSpatialDatabaseRecordList();
        doGeometryTestsOnResults(bbox, results);
        tx.success();
    }
} finally {
    database.shutdown();
}
```

4. 导出地图文件

在前面介绍的例子中导入的 ESRI Shapefile，实际上是由 Neo4j Spatial 创建的。我们还可以利用查询的结果或者使用 DynamicLayer 抽取出一个单一图层，用 ShapefileExporter 导出一个新的 Shapefile。如果要导出通过导入 Shapefile 创建的完整图层，在这里是不太可能的，但是可以使用此功能将图层的子集导出成 Shapefile，或者将用其他格式数据创建的图层导出为 Shapefile。

```
SpatialDatabaseService spatialService = new SpatialDatabaseService(database);
try (Transaction tx = database.beginTx()) {
    OSMLayer layer = (OSMLayer) spatialService.getLayer("map.osm");
    DynamicLayerConfig wayLayer = layer
        .addSimpleDynamicLayer(Constants.GTYPE_LINESTRING);
    ShapefileExporter shpExporter = new ShapefileExporter(database);
    shpExporter.exportLayer(wayLayer.getName());
    tx.success();
}
```

以上示例显示如何导入具有多个不同几何类型数据的 OSM 数据集，然后选择 DynamicLayer 类型为 LineString 的几何形状，采用这种方式可以将所有 OSM 数据导出到 Shapefile，因为 SHP 格式不允许每个 Shapefile 具有多种几何类型。

```
SpatialDatabaseService spatialService = new
```

```
SpatialDatabaseService(database);
Layer layer = spatialService.getLayer("map.osm");
LayerIndexReader spatialIndex = layer.getIndex();
System.out.println("Have " + spatialIndex.count() + " geometries in " +
spatialIndex.getBoundingBox());
Envelope bbox = new Envelope(12.94, 12.96, 56.04, 56.06);
    try (Transaction tx = database.beginTx()) {
      List<SpatialDatabaseRecord> results = GeoPipeline
          .startIntersectWindowSearch(layer, bbox)
          .toSpatialDatabaseRecordList();
      spatialService.createResultsLayer("results", results);
      ShapefileExporter shpExporter = new ShapefileExporter(database);
      shpExporter.exportLayer("results");
      tx.success();
    }
```

这次我们导入相同的 OSM 模型，但查询输入范围之内的所有形状，并将其导出到新的 Shapefile 中。

关于更多的空间索引查询可以查阅 GeoPipes[39]。

8.1.1.7 常用的 Cpyher 空间查询

1. WithinDistance 查询

WithinDistance 查询是空间查询中最常用的一种方式，使用的是球面距离，计算采用 OrthodromicDistance 算法。

Neo4j Spatial 中 OrthodromicDistance 算法的实现为：org.neo4j.gis.spatial.pipes.processing.OrthodromicDistance。返回结果默认以命中目标坐标与查询中心点距离进行排序。

参考 Neo4j Spatial 源码测试用例中的 TestSimplePointLayer 中的 checkPoint Order。如下所示：

```
List<GeoPipeFlow> res = GeoPipeline
.startNearestNeighborLatLonSearch( layer, start, distance)
.sort("OrthodromicDistance").toList();
```

查询点(120.678966, 31.300864)周边 0.1km 范围以内的点：

格式

```
START n = node:<layer>("withinDistance:[<y>, <x>, <max distance in km>]")
```

例子

```
start n = node:geom('withinDistance:[31.331937,120.638154,0.1]') return n
limit 10
```

[39] https://github.com/geopipes

2. bbox 矩形查询

查询由点(120.678966,31.300864)与点(120.978966,31.330864)构成的 BBox 矩形范围内的点：

格式

```
START n = node:<layer>("bbox:[<min x>, <max x>, <min y>, <max y>]")
```

例子

```
start n = node:geom('bbox:[120.678966,120.978966,31.300864, 31.330864]')
return n limit 10
```

3. WithinWKTGeometry 查询

查询由点(120.678966,31.300864)与点(120.978966,31.330864)构成的 Polygon 多边形范围内的点：

格式

```
START n = node:<layer>("withinWKTGeometry:POLYGON((<x1> <y1>, ..., <xN>
<yN>, <x1> <y1>))")
```

例子

```
start n = node:geoindex('withinWKTGeometry:POLYGON ((120.678966 31.300864,
120.678966 31.330864, 120.978966 31.330864, 120.978966 31.300864, 120.678966
31.300864))')  return n limit 10
```

8.1.1.8　使用 Java 添加空间索引和进行查询

定义哪些属性是代表坐标的（在这里是“lat”和“lon”属性）。

```
final Map<String, String> config = new HashMap<String, String>();
config.put(LayerNodeIndex.LAT_PROPERTY_KEY, "lat");
config.put(LayerNodeIndex.LON_PROPERTY_KEY, "lon");
config.put(SpatialIndexProvider.GEOMETRY_TYPE,
LayerNodeIndex.POINT_PARAMETER);
```

创建图层的索引：

```
LayerNodeIndex layerIndex = new LayerNodeIndex("layerIndex", graphDb,
config);
```

将节点添加到索引：

```
layerIndex.add(dbNode, "", "");
```

现在就可以使用一般的索引查询方法来查询给定点的一定距离内的所有点，并且按照距离的升序排序。

```
IndexHits hits  = index.query(LayerNodeIndex.WITHIN_DISTANCE_QUERY, params);
Double coords = new Double[]{ 10.415039d, 51.151786d };
```

```
Map<String, Object> params = new HashMap<String, Object>();
params.put(LayerNodeIndex.POINT_PARAMETER, coords.toArray());
params.put(LayerNodeIndex.DISTANCE_IN_KM_PARAMETER, dist);
IndexHits hits  = index.query(LayerNodeIndex.WITHIN_DISTANCE_QUERY, params);
```

如同其他的基于索引的查询一样，我们可以得到多个结果组成的迭代器（Iterable），在结果中迭代，获得找到的所有点。

```
for (Node spatialNode : hits) {
    Node dbNode = graphDb.getNodeById((Long) spatialNode
    .getProperty("id"));
    }
```

8.1.2 自定义中文全文索引

Neo4j 也提供全文索引机制，并且是基于 Lucene[40]（开源的全文检索引擎框架）实现的，但是默认情况下 Lucene 只提供了基于英文的分词器，如默认的 exact 查询采用的是 Lucene 自带的 KeywordAnalyzer（关键词分词器），fulltext 查询采用的是 white-space tokenizer（空格分词器），如果将这个分词器直接用于中文的分词，会将中文分成单个的字，这样就破坏了中文的语义结构。而且对于特定应用领域的中文分词，可能还需要自定义词典，因此需要在 Neo4j 的全文索引机制上针对中文分词做一些改进。

此外，使用默认的查询不一定能得到我们想要的结果，因此还需要自定义查询结果排序。本小节将介绍 Neo4j 怎么使用自定义中文全文索引。

要实现自定义全文中文索引，首先要解决的就是分词器，中文分词器的种类有很多，在这里就不一一介绍。本节使用的是开源的 IKAnalyzer 分词器。

8.1.2.1 IKAnalyzer 分词器

IKAnalyzer 是一个开源的、基于 Java 语言开发的轻量级中文分词工具包。IKAnalyzer 3.0 的特性如下：

- 采用了特有的“正向迭代最细粒度切分算法”，支持细粒度和最大词长两种切分模式；具有 83 万字/秒（1600KB/S）的高速处理能力。
- 采用了多子处理器分析模式，支持英文字母、数字、中文词汇等分词处理，兼容韩文、日文字符优化的词典存储，内存占用更小，并支持用户词典扩展定义。
- 针对 Lucene 全文检索优化的查询分析器 IKQueryParser，引入简单搜索表达式，采用歧义分析算法优化查询关键字的搜索排列组合，能极大地提高 Lucene 检索的命中率。

然而，IKAnalyser 目前还没有 Maven 库，而且 IKAnalyzer 官方只更新到支持 Lucene3.0 版本，Neo4j 中自带的 Lucene 为 5.5.0 版本，因此想要 IKAanlyzer 支持 Lucene5.5.0 版本还需要做一些改进，具体操作请读者自行查阅一些相关资料。随后，用户可以添加对应的 jar 包到 Maven 本地仓库。

[40] http://lucene.apache.org/

8.1.2.2　自定义用户词典

词典文件：自定义词典后缀名为.dic 的词典文件，必须使用 UTF-8 编码保存。

词典配置：IKAnalyzer.cfg.xml 必须在 src 根目录下。词典可以放置在任意文件路径中，但是在 IKAnalyzer.cfg.xml 里要配置正确。如下的这种配置，ext.dic 和 stopword.dic 应当在同一目录下。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment>IK Analyzer 扩展配置</comment>
    <!--用户可以在这里配置自己的扩展字典 -->
    <entry key="ext_dict">ext.dic;</entry>
    <!--用户可以在这里配置自己的扩展停止词字典-->
    <entry key="ext_stopwords">stopword.dic;</entry>
</properties>
```

8.1.2.3　嵌入式模式下 Neo4j 中文全文索引

指定 IKAnalyzer 作为 Luncene 分词的 Analyzer，并对一个 Label 下的所有 Node 指定属性新建全文索引。

```
try (Transaction tx = graphDBService.beginTx()) {
   private static final Map<String, String> config =
stringMap(IndexManager.PROVIDER, "lucene", "type", "fulltext", "analyzer",
IKAnalyzer.class.getName());
   IndexManager index = graphDBService.index();
   Index<Node> newFullTextIndex =index.forNodes( "FullTextIndex", config);
   //找到图中某个 Label 的所有点
   ResourceIterator<Node> nodes = graphDBService
       .findNodes(DynamicLabel.label(LabelName));
   while (nodes.hasNext()) {
        Node node = nodes.next();
        //为 text 属性添加索引
        Object text = node.getProperty( "text", null);
        newFullTextIndex.add(node, "text", text);
    }
    tx.success();
   }
```

8.1.2.4　嵌入式模式下 Neo4j 全文检索查询

对关键词（如‘数据库’），多关键词模糊查询（如‘中国 图 数据库’）默认都能检索，且检索结果按关联度已排好序。

```
try (Transaction tx = graphDBService.beginTx()) {
   private static final Map<String, String> config =
stringMap(IndexManager.PROVIDER,"lucene", "type", "fulltext", "analyzer",
IKAnalyzer.class.getName());
   IndexManager index = graphDBService.index();
```

```
    Index<Node> chineseFullTextIndex = index.forNodes("FullTextIndex" ,
config);
    IndexHits<Node> foundNodes = newFullTextIndex.query("text" ,
queryString );
    for (Node node : foundNodes) {
          System.out.println(node.get("text"));
    }
```

Cypher 中使用自定义全文索引查询：

```
START b=node:chineseFullTextIndex("text:中文*")
return b limit 10
```

8.1.2.5 服务器模式下 Neo4j 自定义中文全文索引

在服务器模式下，自定义中文全文索引需要用到自定义过程和自定义函数的知识。可以添加中文全文索引，基于中文全文索引的检索实现为过程，然后使用 CALL 直接调用过程即可。

一个自定义过程的实现如下：

```
public class FulltextIndex{
private static final Map<String, String> FULL_INDEX_CONFIG =
stringMap(IndexManager.PROVIDER, "lucene", "type", "fulltext", "analyzer",
"org.wltea.analyzer.lucene.IKAnalyzer");
// 获取 GraphDatabaseService
@Context
public GraphDatabaseService db;
@Context
public Log log;
@Procedure(value = "userdefined.index.chineseFulltextIndexSearch", mode =
Mode.WRITE)
@Description("call userdefined.index.chineseFulltextIndexSearch (String
indexName, String query, long limit) yield node, 执行 lucene 全文搜索，返回前
{limit} 个结果")
public Stream<ChineseHit> search(@Name("indexName") String indexName,
@Name("query") String query, @Name("limit") long limit
){
    if( !db.index().existsForNodes( indexName ))
    {
        // 输出日志
        log.debug( "如果索引不存在就跳过本次查询: `%s`", indexName );
        return Stream.empty();
    }
    return db.index()
            .forNodes(indexName, FULL_INDEX_CONFIG)
        //根据匹配分数排序，sortByScore
            .query(new QueryContext(query).sortByScore() .top((int)limit))
            .stream()
            .map(ChineseHit::new);
}
    @Procedure(value = "userdefined.index.addChineseFulltextIndex",
```

```
mode=Mode.WRITE)
    @Description("call userdefined.index.addChineseFulltextIndex(String
indexName,String labelName, List<String> propKeys,为一个标签下的所有节点的指定属
性添加索引")
    public void addIndex( @Name("indexName") String indexName,
                          @Name("labelName") String labelName,
                          @Name("properties") List<String> propKeys )
    {
        Label label = Label.label(labelName);
        //按照标签找到该标签下的所有节点
        ResourceIterator<Node> nodes = db.findNodes(label);
        while(nodes.hasNext()){
            Node node = nodes.next();
            //每个节点上想要添加索引的属性
            Set<Map.Entry<String,Object>> properties =
                  node.getProperties( propKeys.toArray( new
String[0] ) ).entrySet();
            //查询该节点是否已有索引，有的话删除
            Index<Node> index = db.index().forNodes( indexName,
FULL_INDEX_CONFIG);
            index.remove( node );
            //为该节点的每个需要添加索引的属性添加全文索引
            for ( Map.Entry<String,Object> property : properties )
            {
                index.add( node, property.getKey(), property.getValue() );
            }
        }
    }
    public static class ChineseHit
    {
        public Node node;
        public ChineseHit(Node node) {this.node = node;}
    }
}
```

将该类编译成 jar 包，并且将其放置于 neo4j/plugins 文件夹下，然后重启 Neo4j 服务，即可直接调用如下过程添加自定义中文全文索引和查询的功能。

```
CALL userdefined.index.chineseFulltextIndexSearch(indexName, query, limit)
CALL userdefined.index.addChineseFulltextIndex(indexName, labelName, propKey)
```

8.2 在 Docker 环境下部署 Neo4j

Docker [41]是一个开源的应用容器引擎，让开发者可以打包应用以及依赖包到一个可移植的容器中，然后发布到 Linux 服务器上，也可以实现虚拟化。容器完全使用沙箱机制，相互

[41] https://www.docker.com/

之间不会有任何接口。

本节中我们学习如何在 Docker 环境下部署 Neo4j，关于 Docker 的安装请参考 Docker 官网，在这里不做讨论。需要注意的是，Docker 无法直接运行在 Windows 和 Mac OS X 系统中，要想在 Windows 和 Mac OS X 中安装 Docker 请参考 Docker 安装资料。

8.2.1 概览

默认情况下 Neo4j 的 Docker 镜像开放了三个用于远程访问的端口：

- 7474 用于 HTTP 协议。
- 7473 用于 HTTPS 协议。
- 7687 用于 BOLT 协议。

此外还指定了两个位置用于存放数据文件和日志文件：

- /data，用于数据库在容器的外部持久化。
- /logs，用于允许访问 Neo4j 日志文件。

在 Docker 中运行 Neo4j：

```
docker run
    --publish=7474:7474 --publish=7687:7687
    --volume=$HOME/neo4j/data:/data
    --volume=$HOME/neo4j/logs:/logs
    neo4j:3.1
```

在 Linux 系统中可以直接运行 http://localhost:7474，如果是在 Mac OS X 或者 Windows 的虚拟机中安装的 Docker 可以运行 http://$(Docker-machine):7474。在此设置中，Neo4j 指定的所有文件位置都存储在$HOME 中。



默认情况下，Neo4j 需要身份验证，默认密码为 neo4j/neo4j，并且要求在第一次登录时设置新的密码。可通过 -env NEO4J_AUTH=neo4j/<password> 指令直接设置 Neo4j 容器的登录密码，或者也可使用 -env NEO4J_AUTH=none 来禁用密码。

标签可以用于区别 neo4j 社区版和企业版。企业版采用-enterprise 后缀的方式标明，例如 neo4j:3.1.0-enterprise，而社区版标签没有后缀，例如 neo4j:3.1.0。最新的 neo4j 企业版可用版本可以使用 neo4j:enterprise。

8.2.2 Docker 配置

如果 Neo4j 同时有大量的数据库连接，可能会使用大量的文件描述符。Docker 容器中控制打开文件描述符的数量取决于系统的配置，建议这个数量至少设置为 40000。要覆盖单个容器的默认连接数量，请使用 -ulimit 配置项。

```
docker run
```

```
    --publish=7474:7474 --publish=7687:7687
    --volume=$HOME/neo4j/data:/data
    --volume=$HOME/neo4j/logs:/logs
    neo4j:3.1
```

8.2.3　Neo4j 配置

本小节提供的默认配置仅适用于 Neo4j 学习，要想其适用于实际生产应用还要做一些修改。特别是默认分给 Neo4j 的存储是非常有限的，以允许多个容器可以在一个服务器上运行。

有三种方式可以修改 Neo4j 容器的配置：

- 设置环境变量。
- 指定/conf 存储位置。
- 构建新的镜像。

选择哪一种方式取决于需要多少个自定义镜像。

8.2.3.1　设置环境变量

在运行 Neo4j 容器时将环境变量传递给容器：

```
docker run
    --detach
    --publish=7474:7474 --publish=7687:7687
    --volume=$HOME/neo4j/data:/data
    --volume=$HOME/neo4j/logs:/logs
    --env=NEO4J_dbms_memory_pagecache_size=4G
    neo4j:3.1
```

下面介绍可用的环境变量：

- NEO4J_AUTH：控制身份验证，设置为 none 禁用身份验证或使用 neo4j/<password> 覆盖默认密码。
- NEO4J_dbms_memory_pagecache_size：Neo4j 的本地内存缓存的大小，默认为 512MB。
- NEO4J_dbms_memory_heap_maxSize：Neo4j 的堆大小，默认为 512MB。
- NEO4J_dbms_txLog_rotation_retentionPolicy：保留逻辑日志的大小，默认为 100MB。
- NEO4J_dbms_allowFormatMigration：设置为 true 启用升级，默认为 false。
- NEO4J_dbms_mode：数据库模式，默认为 SINGLE，设置为 CORE 或 READ_REPLICA 用于因果集群，设置为 HA 用于高可用性集群。注意，该设置仅在 Neo4j 企业版中才可以使用。

1. 因果集群设置

- NEO4J_causalClustering_expectedCoreClusterSize：启动时的初始集群大小（核心实

例个数）。

- NEO4J_causalClustering_initialDiscoveryMembers：集群核心成员的初始网络地址/端口号列表。
- NEO4J_causalClustering_discoveryAdvertisedAddress：IP 地址/端口号，用于成员发现。
- NEO4J_causalClustering_transactionAdvertisedAddress：IP 地址/端口号，用于广播事务处理。
- NEO4J_causalClustering_raftAdvertisedAddress：IP 地址/端口号，用于广播集群通信。

2. 高可用性集群设置

- NEO4J_ha_serverId：服务器的唯一标识符，必须是整数，不可重复。
- NEO4J_ha_host_coordination：用于 HA 模式下集群协调通信的地址（包括端口），这必须是可以由所有集群成员解析的。
- NEO4J_ha_host_data：在 HA 模式下用于数据传输的地址（包括端口），这必须是可以由所有集群成员解析的。
- NEO4J_ha_initialHosts：集群的其他成员的地址/端口号列表。

8.2.3.2 /conf 配置文件设置

要想对 Neo4j 配置进行任意修改，请为容器提供/conf 配置文件存储位置。

```
docker run
    --detach
    --publish=7474:7474 --publish=7687:7687
    --volume=$HOME/neo4j/data:/data
    --volume=$HOME/neo4j/logs:/logs
    --volume=$HOME/neo4j/conf:/conf
    neo4j:3.1
```

/conf 中的任何配置文件都将覆盖由镜像提供的文件，包括已经生效的、为容器提供的环境变量。如果要修改文件中的一个值，必须保证文件中的其余部分是正确的。

8.2.3.3 Neo4j 因果集群模式

为了使因果集群在 Docker 下运行 Neo4j，需要 Docker 中的容器互相连接，以便容器之间可以互相通信。每个容器必须有一个与其他容器通信的网络路由，核心实例必须要设置这两个参数：NEO4J_causalClustering_expectedCoreClusterSize 和 NEO4J_causalClustering_initialDiscoveryMembers。只读副本只需要设置 NEO4J_causalClustering_initialDiscoveryMembers。

在单个 Docker 实例中可以这样实现：

```
docker network create --driver=bridge cluster
docker run --name=core1 --detach --network=cluster
       --publish=7474:7474 --publish=7687:7687
       --env=NEO4J_dbms_mode=CORE
       --env=NEO4J_causalClustering_expectedCoreClusterSize=3
```

```
        --env=NEO4J_causalClustering_initialDiscoveryMembers=
        core1:5000,core2:5000,core3:5000
        neo4j:3.1-enterprise
docker run --name=core2 --detach --network=cluster
        --env=NEO4J_dbms_mode=CORE
        --env=NEO4J_causalClustering_expectedCoreClusterSize=3
        --env=NEO4J_causalClustering_initialDiscoveryMembers=
        core1:5000,core2:5000,core3:5000
        neo4j:3.1-enterprise
docker run --name=core3 --detach --network=cluster
        --env=NEO4J_dbms_mode=CORE
        --env=NEO4J_causalClustering_expectedCoreClusterSize=3
        --env=NEO4J_causalClustering_initialDiscoveryMembers=
        core1:5000,core2:5000,core3:5000
        neo4j:3.1-enterprise
```

其他实例可以由 ad-hoc 方式添加到集群。例如，可用如下方式添加只读副本。

```
docker run --name=read_replica1 --detach --network=cluster
        --env=NEO4J_dbms_mode=READ_REPLICA
        --env=NEO4J_causalClustering_initialDiscoveryMembers=
        core1:5000,core2:5000,core3:5000
        neo4j:3.1-enterprise
```

当每个容器在自己独立的物理机上运行并且不使用 Docker 内部网络的时候，有必要定义每个机器的广播地址，以保证各个物理机之间的通信。每个实例将会被如下方式调用。

```
docker run --name=neo4j-core --detach
        --publish=7474:7474 --publish=7687:7687
        --publish=5000:5000 --publish=7000:7000
        --env=NEO4J_dbms_mode=CORE
        --env=NEO4J_causalClustering_expectedCoreClusterSize=3
       --env=NEO4J_causalClustering_initialDiscoveryMembers=<core1-public-
address>:5000,<core2-public-address>:5000,<core3-public-address>:5000
        --env=NEO4J_causalClustering_discoveryAdvertisedAddress=
        <public-address>:5000
        --env=NEO4J_causalClustering_transactionAdvertisedAddress=
        <public-address>:6000
        --env=NEO4J_causalClustering_raftAdvertisedAddress=
        <public-address>:7000
        neo4j:3.1-enterprise
```

其中<public-address>是每台物理机的 IP 地址。

8.2.3.4　Neo4j 高可用集群模式

以高可用集群的模式在 Docker 上运行 Neo4j，每个容器必须具有可以连接到其他容器的网络路由 NEO4J_ha_host_coordination，还有 NEO4J_ha_host_data 和 NEO4J_ha_initialHosts。

在单个 Neo4j 实例中，可以做如下配置：

```
docker network create --driver=bridge cluster
docker run --name=instance1 --detach --publish=7474:7474 --publish=7687:7687
--net=cluster --hostname=instance1
    --volume=$HOME/neo4j/logs1:/logs
    --env=NEO4J_dbms_mode=HA --env=NEO4J_ha_serverId=1
    --env=NEO4J_ha_host_coordination=instance1:5001
    --env=NEO4J_ha_host_data=instance1:6001
--env=NEO4J_ha_initialHosts=instance1:5001,instance2:5001,
instance3:5001
    neo4j:3.1-enterprise
docker run --name=instance2 --detach --publish 7475:7474 --publish=7688:7687
--net=cluster --hostname=instance2
    --volume=$HOME/neo4j/logs2:/logs
    --env=NEO4J_dbms_mode=HA --env=NEO4J_ha_serverId=2
    --env=NEO4J_ha_host_coordination=instance2:5001
    --env=NEO4J_ha_host_data=instance2:6001
--env=NEO4J_ha_initialHosts=instance1:5001,instance2:5001,
instance3:5001
    neo4j:3.1-enterprise
docker run --name=instance3 --detach --publish 7476:7474 --publish=7689:7687
--net=cluster --hostname=instance3
    --volume=$HOME/neo4j/logs3:/logs
    --env=NEO4J_dbms_mode=HA --env=NEO4J_ha_serverId=3
    --env=NEO4J_ha_host_coordination=instance3:5001
    --env=NEO4J_ha_host_data=instance3:6001
--env=NEO4J_ha_initialHosts=instance1:5001,instance2:5001,
instance3:5001
    neo4j:3.1-enterprise
```

8.2.3.5 使用用户自定义过程

要使用用户自定义过程，需要指定/plugins 的位置。

```
docker run
    --publish 7474:7474
    --publish=7687:7687
    --volume=$HOME/neo4j/plugins:/plugins
    neo4j:3.1
```

8.2.3.6 使用 Cypher-Shell

Cypher-Shell 可以在容器内使用以下命令在本地运行：

```
docker exec --interactive --tty <container> bin/cypher-shell
```

8.3 自定义批量导入

往数据库中导入大量的数据，为了保证导入的速度，一般都需要批量处理。Neo4j 提供

了批量导入工具 neo4j-import 和 batch-import，但不论是 batch-import 工具还是 neo4j-batch 工具都具有一定的局限性。如果想要对特殊格式的数据直接进行批量导入，而不是转换成特定格式的 csv 文件，可以写一个自定义的批量导入工具。

Neo4j 提供了 BatchInserter 接口用来批量导入数据，但要注意的是使用 BatchInserter 是不安全的，因为 BatchInserter 没有使用事务机制。使用 BatchInserter 时请注意以下几点：

- 批量导入一般在初始导入数据时使用，如果数据库中已经有数据的情况，需要事先关闭数据，然后才可以正常使用。
- 批量导入不是线程安全的。
- 批量导入是非事务的。
- 批量导入不会在插入数据时对插入的数据强制实施约束。
- 当批量导入关闭（Shutdown）时，所有已经存在的索引和批量导入期间新加入的索引都会重新生成。
- 当批量导入关闭（Shutdown）时，所有已经存在的约束和在批量导入期间新加入的约束都会被重新确认。
- 如果批量导入没有被正常关闭（Shutdown），则数据库会出错。

下面通过一个简单的批量导入例子来说明 BatchInserter 的使用。

```
BatchInserter inserter = null;
try
{
    inserter = BatchInserters.inserter(
    new File( "target/batchinserter-example" ).getAbsolutePath() );
    Label personLabel = DynamicLabel.label( "Person" );
    inserter.createDeferredSchemaIndex( personLabel ).on( "name" ).create();
    Map<String, Object> properties = new HashMap<>();
    properties.put( "name", "Mattias" );
    long mattiasNode = inserter.createNode( properties, personLabel );
    properties.put( "name", "Chris" );
    long chrisNode = inserter.createNode( properties, personLabel );
    RelationshipType knows = DynamicRelationshipType.withName( "KNOWS" );
    inserter.createRelationship( mattiasNode, chrisNode, knows, null );
}
finally
{
    if ( inserter != null )
    {
        inserter.shutdown();
    }
}
```

在批量导入的过程中，还可以根据自己机器的性能设置批量导入的各方面性能。这里有两种方式设置批量导入配置。一种是直接将配置项写在一个 Map 中，例如：

```
Map<String, String> config = new HashMap<>();
config.put( "dbms.pagecache.memory", "512m" );
```

```
BatchInserter inserter = BatchInserters.inserter( new
File( "filePath" ).getAbsolutePath(), config );
// 在这里插入数据，然后 shutdown
inserter.shutdown();
```

另一种是将配置项单独的写在一个配置文件中。可以写如下配置文件，并且命名为batchinserter.properties。

```
dbms.pagecache.memory=8m
```

然后用下面方式读取配置文件：

```
FileReader input = new FileReader( new
File( "target/batchinserter.properties " ).getAbsoluteFile());
{
    Map<String, String> config = MapUtil.load( input );
    BatchInserter inserter = BatchInserters.inserter(
          new File( "filePath" ).getAbsolutePath(), config );
    //在这里插入数据，然后 shutdown
    inserter.shutdown();
}
```

批量导入配置项的设置与用户本机配置有关，如果 Java 堆的最大值小于 4GB，那么推荐使用如下配置项：

```
cache_type = none
use_memory_mapped_buffers = true
neostore.nodestore.db.mapped_memory = 200M
neostore.relationshipstore.db.mapped_memory = 2G
neostore.propertystore.db.mapped_memory = 200M
neostore.propertystore.db.strings.mapped_memory = 200M
neostore.propertystore.db.index.keys.mapped_memory = 5M
neostore.propertystore.db.index.mapped_memory = 5M
```

如果 Java 堆的最大值大于 4GB，那么推荐使用另一个配置项：

```
cache_type = none
use_memory_mapped_buffers = true
neostore.nodestore.db.mapped_memory = 1G
neostore.relationshipstore.db.mapped_memory = 3G
neostore.propertystore.db.mapped_memory = 1G
neostore.propertystore.db.strings.mapped_memory = 500M
neostore.propertystore.db.index.keys.mapped_memory = 5M
neostore.propertystore.db.index.mapped_memory = 5M
```

如果想在批量导入期间手动添加索引，Neo4j 也提供了 BatchInserterIndex 和 BatchInserterProvider 接口。该索引是由 Lucene 实现的，因此可以在这里添加精确索引，也可以添加全文索引，关键词分别是“exact”和“fulltext”。

```
BatchInserter inserter = BatchInserters.inserter( "filePath", config );
BatchInserterIndexProvider indexProvider =
      new  LuceneBatchInserterIndexProvider( inserter );
```

```
//建立名字为 actor 的索引
BatchInserterIndex actors =
        indexProvider.nodeIndex( "actors", MapUtil.stringMap( "type",
"exact" ) );
//设置索引的缓存能力
actors.setCacheCapacity( "name", 100000 );
Map<String, Object> properties = MapUtil.map( "name", "Keanu Reeves" );
long node = inserter.createNode( properties );
//为节点添加索引
actors.add( node, properties );
//使添加的索引变的可用
actors.flush();
// 需要关闭 index provider
indexProvider.shutdown();
inserter.shutdown();
```

关于使用 BatchInserterIndex 的一些建议如下：

- 避免频繁使用 flush 操作，因为每次使用 flush 之后，新添加的索引就会对查询方法生效，这个操作会有大量的 IO 开销，将会导致性能有不同程度的损失。
- 不要在导入数据结束的阶段忘了做 flush 操作，以免索引不能够生效。
- 可以为以后经常需要查询的数据添加索引缓存，以便将来提高查询的效率。

8.4 Neo4j 与图计算

随着现实世界中网络数据的爆炸式增加，图计算逐渐成为近些年的热点研究问题之一。Neo4j 作为图数据库方面的翘楚，也在图计算问题上做了很大的努力。

本节将简单地介绍如何有效地使用 Neo4j 与 Apache Spark，在这里将列出一些简单的方法，指导利用 Neo4j 和 Spark 建立自己的图计算解决方案。

在开始本节之前，应该对 Apache Spark 和 Neo4j 的数据模型、数据范例和 API 有一个较好的了解，以利于后续学习。

Apache Spark 是一种集群内数据处理的解决方案，可以轻松地在多个机器上进行大规模数据处理，还带有 GraphX 和 GraphFrames 两个框架，可以专门用于对数据进行图计算操作。

Spark 与 Neo4j 结合可以作为 Neo4j 的外部数据处理解决方案，简单的处理过程如下所示。

（1）所需要分析的子图从 Neo4j 导出到 Spark。
（2）利用 Spark 集群进行图计算。
（3）将计算结果返回到 Neo4j 中。
（4）用 Neo4j 的 Cypher 语言或者其他操作工具进行查询。



Neo4j 本身已经能够在比较大的数据量上进行图计算操作。我们测试过利用 Neo4j Pagerank 过程可以在 20 秒内对 10MB 节点，25MB 关系运行 PageRank 算法（5 次迭代）。Spark 适合处理更大的数据集和更密集的操作。

8.4.1 Neo4j-Spark-Connector

Neo4j-Spark-Connector 使用二进制 Bolt 协议从 Neo4j 中导入和导出数据。Neo4j-Spark-Connector 提供了 Spark-2.0 的 RDD、DataFrame、GraphX Graph 和 GraphFrames 等 API，可以自由地选择怎样使用 Spark 处理 Neo4j 数据。

一般的分析过程如下：

（1）创建 org.neo4j.spark.Neo4j(sc)。

（2）设置 cypher(query, [params]), nodes(query, [params]), rels(query, [params])作为直接查询或者 pattern("Label1", Seq("REL"), "Label2") 或 pattern(("Label1", "prop1"), ("REL", "prop"), ("Label2", "prop2"))。

（3）为并行计算定义 partitions(n)，batch(size)，rows(count)。

（4）选择返回的数据类型。

- loadRowRdd、loadNodeRdds、loadRelRdd、loadRdd[T]。
- loadDataFrame、loadDataFrame(schema)。
- loadGraph[VD,ED]。
- loadGraphFrame[VD,ED]。

下面实现一个简单地从 Neo4j 中加载数据并且使用 Spark 分析的流程。在开始这个教程之前，需要具备 Spark 和 Scala 的相关知识，并且需要安装配置 Spark 和 Scala 的运行环境。

为了完成整个处理流程的描述，在这里需要向 Neo4j 数据库中添加数据。添加数据的 Cypher 语句如下所示。

```
UNWIND range(1,100) as id
CREATE (p:Person {id:id}) WITH collect(p) as people
UNWIND people as p1
UNWIND range(1,10) as friend
WITH p1, people[(p1.id + friend) % size(people)] as p2
CREATE (p1)-[:KNOWS {years: abs(p2.id - p1.id)}]->(p2)
```

如果使用 Spark-Shell 进行分析，首先需要在 Spark 中配置 Neo4j 的连接方式。

（1）如果使用默认的主机和端口运行 Neo4j，只需要在 spark-default.conf 中配置用户的密码：

```
spark.neo4j.bolt.password = your_neo4j_password
```

（2）如果运行的不是 Neo4j 默认的主机和端口，那么需要做如下配置：

```
spark.neo4j.bolt.url=bolt://$host_name:$port
spark.neo4j.bolt.user= your_neo4j_username
spark.neo4j.bolt.password = your_neo4j_password
```

（3）第二种设置方式也可以用另一种方式给出：

```
spark.neo4j.bolt.url =bolt://neo4j:<password>@$host_name: $port
```

然后可以运行 Spark-Shell，打开 Spark-Shell 的时候还要添加 Neo4j-Spark-Connector 的

jar 包依赖。可以用如下方式添加：

```
$SPARK_HOME/bin/spark-shell --packages neo4j-contrib: neo4j-spark-
connector:2.0.0-M2
```

也可以将 Neo4j-Spark-Connector 下载到本地，用如下方式添加：

```
$SPARK_HOME/bin/spark-shell --jars neo4j-spark-connector_2.11 -full-2.0.0-
M2.jar
```

如果还要添加其他 jar 包依赖，直接在上述 jar 包后面添加依赖的 jar 包名称即可，并且用逗号隔开，例如：

```
$SPARK_HOME/bin/spark-shell --packages neo4j-contrib:neo4j-spark -
connector:2.0.0-M2,graphframes:graphframes:0.2.0-spark2.0-s_2.11
```

首先从 Neo4j 中加载数据并且转化成 Spark 的 RDD 类型，在 Spark-Shell 中运行如下代码，并且对照输出结果：

```
import org.neo4j.spark._
val neo = Neo4j(sc)
val rdd = neo.cypher("MATCH (n:Person) RETURN id(n) as id ").loadRowRdd
rdd.count
//   => Long = 100
rdd.first.schema.fieldNames
//   => ["id"]
rdd.first.schema("id")
//   => StructField(id,LongType,true)
neo.cypher("MATCH (n:Person) RETURN id(n)").loadRdd[Long].mean
//   => Double = 236696.5
neo.cypher("MATCH (n:Person) WHERE n.id <= {maxId} RETURN
n.id").param("maxId", 10).loadRowRdd.count
//   => Long = 10
// 设置分区和批处理大小
neo.nodes("MATCH (n:Person) RETURN id(n) SKIP {_skip} LIMIT
{_limit}").partitions(4).batch(25).loadRowRdd.count
//   => 100 == 4 * 25
// 通过 pattern 加载数据
neo.pattern("Person",Seq("KNOWS"),"Person").rows(80).batch(21).loadNodeRdds.c
ount
//   => 80
// 通过 pattern 加载关系
neo.pattern("Person",Seq("KNOWS"),"Person").partitions(12).batch(100).loadRel
Rdd.count
//   => 1000
```

从 Neo4j 中加载数据并且转化成 Spark 的 DataFrame 类型，在 Spark-Shell 中运行如下代码，并且对照输出结果：

```
import org.neo4j.spark._
val neo = Neo4j(sc)
// 通过 Cypher 查询加载数据
```

```
neo.cypher("MATCH (n:Person) RETURN id(n) as id SKIP {_skip} LIMIT
{_limit}").partitions(4).batch(25).loadDataFrame.count
//   => res36: Long = 100
val df = neo.pattern("Person",Seq("KNOWS"),"Person").partitions
(12).batch(100).loadDataFrame
//   => org.apache.spark.sql.DataFrame = [id: bigint]
```

从 Neo4j 中加载数据并且转化成 Spark 的 GraphX Graph 类型，在 Spark-Shell 中运行如下代码，并且对照输出结果：

```
import org.neo4j.spark._
import org.apache.spark.graphx._
import org.apache.spark.graphx.lib._
val neo = Neo4j(sc)
// 通过 Cypher 查询加载 graph
val graphQuery = "MATCH (n:Person)-[r:KNOWS]->(m:Person) RETURN id(n) as
source, id(m) as target, type(r) as value SKIP {_skip} LIMIT {_limit}"
val graph: Graph[Long, String] =
neo.rels(graphQuery).partitions(7).batch(200).loadGraph
graph.vertices.count
//    => 100
graph.edges.count
//    => 1000
// 通过 pattern 加载 graph
val graph = neo.pattern(("Person","id"),("KNOWS","since"),("Person","
id")).partitions(7).batch(200).loadGraph[Long,Long]
val graph2 = PageRank.run(graph, 5)
graph2.vertices.sort(_._2).take(3)
```

从 Neo4j 中加载数据并且转化成 Spark 的 GraFrames 类型，在 Spark-Shell 中运行如下代码，并且对照输出结果：

```
import org.neo4j.spark._
import org.graphframes._
val neo = Neo4j(sc)
val graphFrame = neo.pattern(("Person","id"),("KNOWS",null),
("Person","id")).partitions(3).rows(1000).loadGraphFrame
graphFrame.vertices.count
//    => 100
graphFrame.edges.count
//    => 1000
val pageRankFrame = graphFrame.pageRank.maxIter(5).run()
val ranked = pageRankFrame.vertices
ranked.printSchema()
val top3 = ranked.orderBy(ranked.col("pagerank").desc).take(3)
```

8.4.2 Neo4j-Spark-Connector 提供的 API

Neo4j-Spark-Connector 还提供了更简单的加载和操作 RDD、DataFrames、GraphX Graphs 和 Graph Frames 方法。详情如下所示。

1. RDD

- Neo4jTupleRDD：每行返回一个 Seq[(String, AnyRef)]。
- Neo4jRowRDD：每行返回一个 SparkSQL ROW。

2. DataFrame

- 返回 SparkSQL 的 DataFrame。

3. GraphX-Neo4jGraph

- Neo4jGraph：具有加载和保存 GraphX Graph 的方法。
- Neo4jGraph.execute：执行 Cypher 查询并且返回 CypherResult。
- Neo4jGraph.loadGraph(sc, label, rel-types, label2)：通过标签标记的节点之间的关系加载子图。
- Neo4jGraph.saveGraph(g, nodeProp, relProp)：将更新过的节点和关系属性保存到 Neo4j。
- Neo4jGraph.loadGraphFromNodePairs(sc, stmt, params)：从节点 node-id 对中加载图。
- Neo4jGraph.loadGraphFromRels(sc, stmt, params)：从起始节点 id 和结束节点 id 以及它们之间关系的限制加载图。
- Neo4jGraph.loadGraph(sc, (stmt, params), (stmt, params))：加载两个条件约束的图，第一个条件约束点的类型，第二个条件约束边的类型。

4. GraphFrames

- Neo4jGraphFrame(sqlContext,(srcNodeLabel,nodeProp),(relType,relProp),dst:(dstNodeLabel,dstNodeProp)：加载具有给定源节点和目标节点及他们之间关系的图。
- Neo4jGraphFrame.fromGraphX(sc, label, Seq(rel-type), label)：加载具有给定模式的图。
- Neo4jGraphFrame.fromEdges(sqlContext,srcNodeLabel, Seq(relType), dstNodeLabel)：与上一条类似。

以上初步介绍了 Neo4j 如何与 Spark 结合进行图分析，如果要构建你自己的图计算平台，还需要进一步学习，更详细的信息可以参考 https://github.com/neo4j-contrib/neo4j-spark-connector。

8.5 Neo4j 与自然语言处理

自然语言处理技术在挖掘文本数据时使用的关键技术之一是本体的挖掘词关联。词关联在语音处理标记、解析、实体提取等自然语言处理任务中非常有用。Neo4j 由于其强大的处理关联数据的能力，为自然语言处理中词关联的处理提供了一种新的解决方案。在本节中，我们简要介绍称为聚合关系（Paradigmatic Relation）的一种词关联，并且演示如何使用

Neo4j 帮助我们将文本语料库构建成一个图，并实现一种简单的聚合关系挖掘算法。

在自然语言处理中有两种常见的词关联：聚合关系和组合关系（Syntagmatic Relation）。

- 聚合关系：词 A 和词 B 如果彼此之间可以相互替换，就称之为具有聚合关系的，这表示它们属于同一类，例如“星期一”和“星期四”，“猫”和“狗”。
- 组合关系：表示可以相互组合的词，例如“冷”和“天气”。

在这里我们只关心聚合关系。

8.5.1 计算聚合相关性

计算聚合关系的基本步骤分为三步：

1. 通过一个单词的上下文表示每个单词

考虑一个简单的文档，里面包括这样的句子：

```
My cat eats fish on Saturday.
His dog eats turkey on Tuesday.
```

要分析两个词是否具有聚合关系，需要用一些方法来表示给定单词的上下文。我们通过如下函数来计算给定单词的上下文片段：

```
Right1("cat") = {"eats", "ate", "is", "has", ...}
Left1("cat") = {"my", "his", "big", "a", "the", ...}
```

2. 计算上下文相似性

为了计算聚合关系的度量，采用相对上下文相似性的和来表示：

```
Sim("Cat","Dog")=Sim(Left1("Cat"),Left1("Dog"))+Sim(Right1("Cat"), Right1("Dog"))
```

我们可以使用 Jaccard 指数作为相似性的度量。

3. 具有上下文高度相似性的词可能具有聚合关系

一旦有了这种计算相似度的方式，就可以寻找具有高相似度的词对。这是一种相对简单的方法，可以通过扩大上下文窗口的大小，处理停止词来调整相似性得分。

8.5.2 将文本数据建模为邻接图

可以将文本数据建模为邻接图，其中每个单词是一个节点，两个节点之间的边表示这些单词在文本语料库中彼此相邻。我们将使用的数据模型如图 8-3 所示。

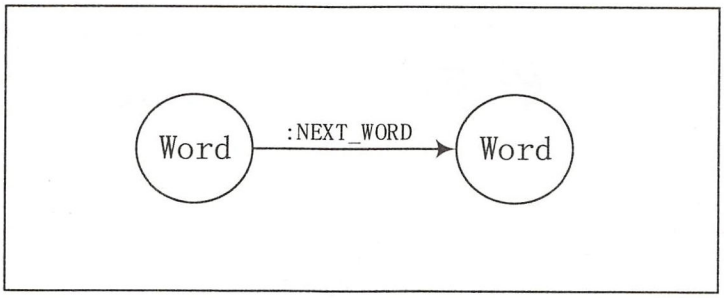


图 8-3　Neo4j 中的词关联模型

考虑如下句子：

```
My cat eats fish on Saturday.
```

使用我们的数据模型，这将在我们的邻接图中显示（如图 8-4 所示）。

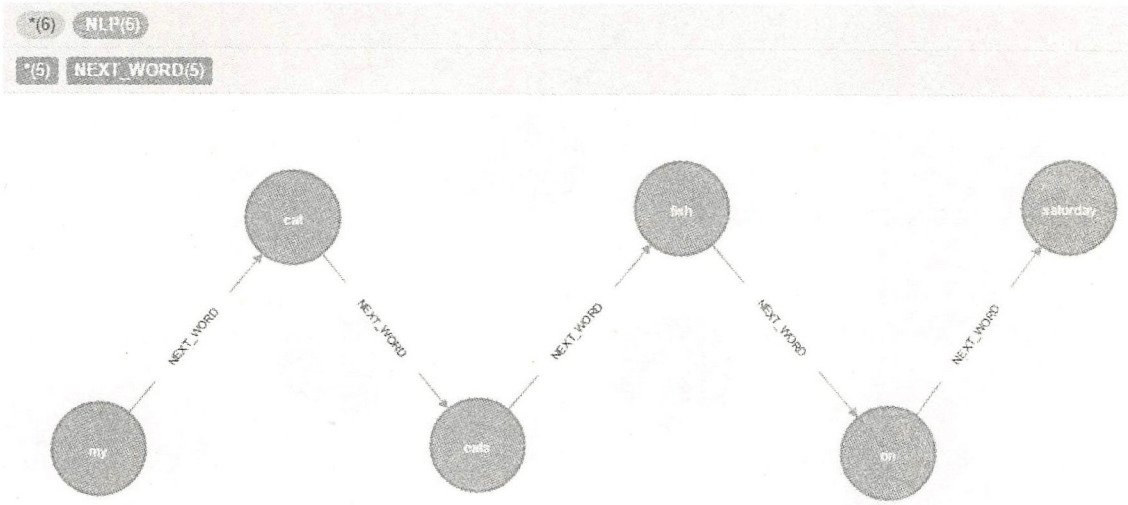


图 8-4　一个句子生成的邻接图

如果添加另一句话：

His dog eats turkey on Tuesday.

更新后的结构如图 8-5 所示。

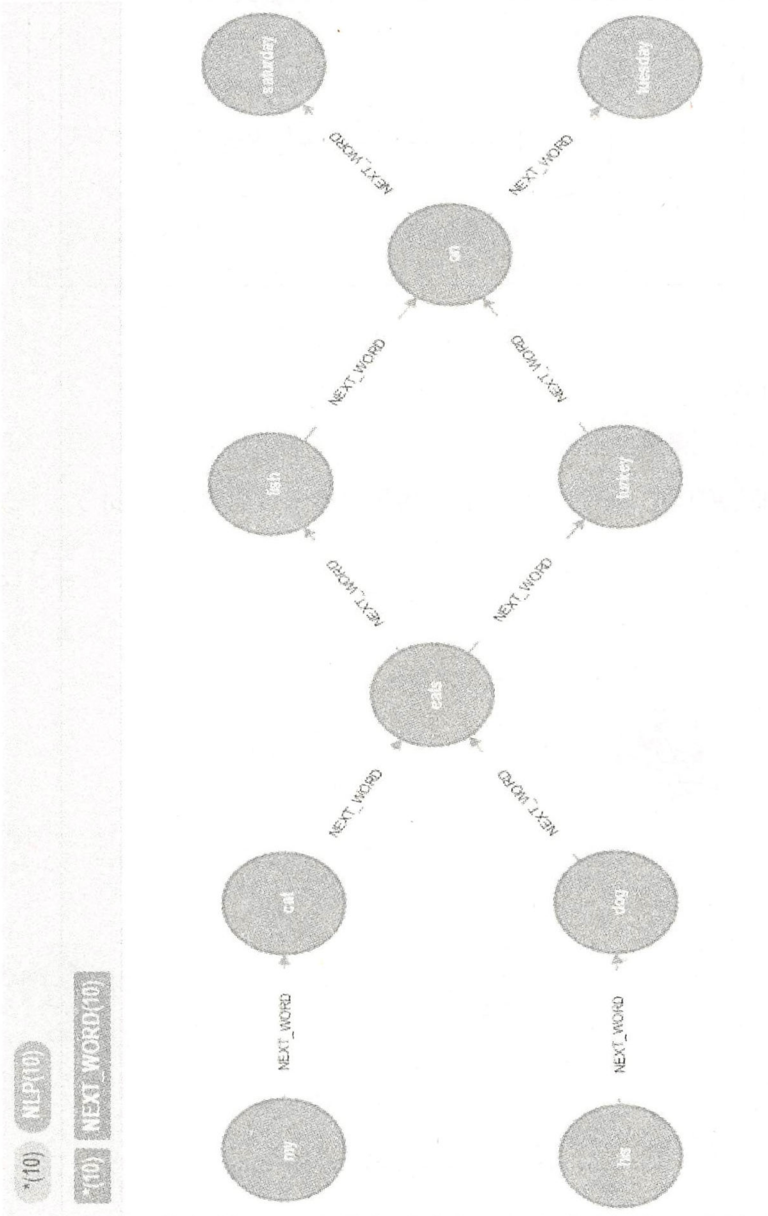


图 8-5　多个句子生成的邻接图

我们将使用 Neo4j 图数据库来实现这个数据模型，构建一个完整的语料库，样例中使用的是 CEEAUS 语料库。

我们将首先看看如何使用 Python 脚本将这个文本语料库加载到 Neo4j 中。然后，将探讨如何使用 Cypher（Neo4j 查询语言）查询数据以挖掘聚合关系。

8.5.3 加载数据

数据存储在单个文件中，每个句子在一行上。我们定义一个 Python 函数来加载这个文

件，对文本进行标准化，并对每行文本执行 Cypher 查询，再根据上面定义的数据模型插入到图中。这里是用于数据插入的 Python 代码：

```
from py2neo import Graph
import re, string
# 连接到 Neo4j 的默认 uri
graphdb = Graph('http://neo4j:neo4j@localhost:7474/db/data')
# 插入数据的 cypher 语句
INSERT_QUERY = '''
    FOREACH (t IN {wordPairs} |
        MERGE (w0:Word {word: t[0]})
        MERGE (w1:Word {word: t[1]})
        CREATE (w0)-[:NEXT_WORD]->(w1)
        )
'''
# 建立单词对
# arrifySentence("Hi there, Bob!) = [["hi", "there"], ["there", "bob"]]
def arrifySentence(sentence):
    sentence = sentence.lower()
    sentence = sentence.strip()
    exclude = set(string.punctuation)
    regex = re.compile('[%s]' % re.escape(string.punctuation))
    sentence = regex.sub('', sentence)
    wordArray = sentence.split()
    tupleList = []
    for i, word in enumerate(wordArray):
        if i+1 == len(wordArray):
            break
        tupleList.append([word, wordArray[i+1]])
    return tupleList
# 将我们的语料库数据加载到 Neo4j
def loadFile():
    tx = graphdb.cypher.begin()
    with open('data/ceeaus.dat', encoding='ISO-8859-1') as f:
        count = 0
        for l in f:
            params = {'wordPairs': arrifySentence(l)}
            tx.append(INSERT_QUERY, params)
            tx.process()
            count += 1
            # 每100个插入语句提交一次
            if count > 100:
                tx.commit()
                tx = graphdb.cypher.begin()
                count = 0
    f.close()
    tx.commit()
```

8.5.4 挖掘单词之间的关系

现在已经插入了我们的数据，需要一些方法来查询它从而发现单词之间的关系。

1. Cypher 和 Python 结合的方法

我们可以写一个简单的 Cypher 查询来查找 Right1 和 Left1 的集合，还可以定义一些 Python 方法来处理这些 Cypher 查询并执行一些简单的聚合运算来计算单词对的 Jaccard 相似性：

```
# 得到查询单词语料库中左侧的单词集合
LEFT1_QUERY = '''
    MATCH (s:Word {word: {word}})
    MATCH (w:Word)-[:NEXT_WORD]->(s)
    RETURN w.word as word
'''
# 得到查询单词语料库中右侧的单词集合
RIGHT1_QUERY = '''
    MATCH (s:Word {word: {word}})
    MATCH (w:Word)<-[:NEXT_WORD]-(s)
    RETURN w.word as word
'''
# 返回出现在`word`左侧所有单词的集合
def left1(word):
    params = {
        'word': word.lower()
    }
    tx = graphdb.cypher.begin()
    tx.append(LEFT1_QUERY, params)
    results = tx.commit()
    words = []
    for result in results:
        for line in result:
            words.append(line.word)
    return set(words)
# 返回出现在`word`右侧所有单词的集合
def right1(word):
    params = {
        'word': word.lower()
    }
    tx = graphdb.cypher.begin()
    tx.append(RIGHT1_QUERY, params)
    results = tx.commit()
    words = []
    for result in results:
        for line in result:
            words.append(line.word)
    return set(words)
# 计算 Jaccard 系数
def jaccard(a,b):
```

```
    intSize = len(a.intersection(b))
    unionSize = len(a.union(b))
    return intSize / unionSize
# 聚合相似性为要计算单词的`left1`和`right1`集合的 Jaccard 系数平均值
def paradigSimilarity(w1, w2):
    return (jaccard(left1(w1), left1(w2)) + jaccard(right1(w1), right1(w2)))
/ 2.0
```

通过以上程序我们可以计算任意两个单词之间的相似性。

```
In [174]: paradigSimilarity("school", "university")
Out[174]: 0.2153846153846154
```

2. 纯 Cypher 方法

上面的 Cypher&Python 方法还没办法找到最相似的两个词。我们只能找到 x 和 y 之间的聚合相似性是多少。然而，真正想要知道的可能是与 x 最相似的是哪个词，接下来我们将使用纯 Cypher 方法比较单词对并添加关系：RELATED_TO 更新图结构。这个关系将存储相似性得分。一旦这个查询运行，我们可以使用这个新的关系查询单词之间的联系。

```
MATCH (s:Word)
// 首先得到 left1和 right1
MATCH (w:Word)-[:NEXT_WORD]->(s)
WITH collect(DISTINCT w.word) as left1, s
MATCH (w:Word)<-[:NEXT_WORD]-(s)
WITH left1, s, collect(DISTINCT w.word) as right1
// 匹配除了 s 的其它单词
MATCH (o:Word) WHERE NOT s = o
WITH left1, right1, s, o
// 得到它们的 left1和 right1集合
MATCH (w:Word)-[:NEXT_WORD]->(o)
WITH collect(DISTINCT w.word) as left1_o, s, o, right1, left1
MATCH (w:Word)<-[:NEXT_WORD]-(o)
WITH left1_o, s, o, right1, left1, collect(DISTINCT w.word) as right1_o
// 计算 right1的联合（union）与交集（intersect）
WITH FILTER(x IN right1 WHERE x IN right1_o) as r1_intersect,
  (right1 + right1_o) AS r1_union, s, o, right1, left1, right1_o, left1_o
// 计算 left1的联合（union）和交集（intersect）
WITH FILTER(x IN left1 WHERE x IN left1_o) as l1_intersect,
  (left1 + left1_o) AS l1_union, r1_intersect, r1_union, s, o
WITH DISTINCT r1_union as r1_union, l1_union as l1_union, r1_intersect,
l1_intersect, s, o
WITH 1.0*length(r1_intersect) / length(r1_union) as r1_jaccard,
  1.0*length(l1_intersect) / length(l1_union) as l1_jaccard,
  s, o
WITH s, o, r1_jaccard, l1_jaccard, r1_jaccard + l1_jaccard as sim
CREATE UNIQUE (s)-[r:RELATED_TO]->(o) SET r.paradig = sim;
```

迭代单词对，获取 right1 和 left1 单词集合，然后执行并集（Union）和交集（Intersect）运算，这可以获得每个单词对的 Jaccard 系数。最后，将得到的 Jaccard 系数当作属性存储到

新建立的关系中。

现在我们已经用这些单词关联系数更新了图，可以编写简单的 Cypher 查询来发现单词关联。

与“school”这个词最强烈相关的词是什么？

```
MATCH (s:Word {word: 'school'} )-[r:RELATED_TO]->(o) RETURN
s.word,o.word,r.paradig as sim ORDER BY sim DESC LIMIT 25;
```

运行 Cypher 语句后得到如下结果：

```
+----------+-------------+--------------------+
| s.word   | o.word      | sim                |
+----------+-------------+--------------------+
| "school" | "university" | 0.35416666666666663 |
| "school" | "studies"    | 0.35279106858054227 |
| "school" | "working"    | 0.3497129735935706  |
| "school" | "parents"    | 0.33613445378151263 |
| "school" | "college"    | 0.33519813519813524 |
| "school" | "society"    | 0.3310029130253849  |
| "school" | "parttime"   | 0.3220510229856024  |
| "school" | "study"      | 0.3186426358772177  |
| "school" | "money"      | 0.31618037135278515 |
| "school" | "eating"     | 0.31335453100158983 |
| "school" | "place"      | 0.3115313895455739  |
| "school" | "experience" | 0.30545975141537063 |
| "school" | "work"       | 0.3031372549019608  |
| "school" | "life"       | 0.30304354157565166 |
| "school" | "smoker"     | 0.29819694868238555 |
| "school" | "jobs"       | 0.2965686274509804  |
| "school" | "living"     | 0.29284676833696444 |
| "school" | "future"     | 0.2874499332443258  |
| "school" | "hard"       | 0.28233938346297893 |
| "school" | "children"   | 0.2812025508654722  |
| "school" | "studying"   | 0.27817343946376205 |
| "school" | "food"       | 0.27809596538118875 |
| "school" | "restaurant" | 0.2743995912110373  |
| "school" | "meal"       | 0.2705982542048116  |
| "school" | "friends"    | 0.2692307692307693  |
+----------+-------------+--------------------+
25 rows
45 ms
```

8.6 其他高级应用

Neo4j 的高级应用远不止上面介绍的这些内容，但是限于本章的篇幅在这里就不一一展开，读者可以参考我们给出如下的一些链接做进一步的学习。

Neo4j Framework 模块的链接：https://github.com/graphaware/neo4j-framework/。

Neo4j 推荐引擎构建的链接：https://github.com/graphaware/neo4j-reco。

Neo4j 实时运行模块的链接：https://github.com/graphaware/neo4j-noderank。

Neo4j 与 Elasticsearch 结合的链接：https://github.com/graphaware/neo4j-to-elasticsearch/。

第 9 章

◂Neo4j 简体中文版▸

通过前面几章的学习，我们已经了解 Neo4j 是一种擅长于处理复杂数据关系的新型数据库系统，在各行各业有着广泛的应用前景。但是，要在中国更好地推广和应用，需要解决软件的本地化问题。

本章将全方位介绍由微云数聚（北京）有限公司自主研发实现的 Neo4j 简体中文版。该版本除了提供 Neo4j 简体中文界面外，还提供了特别适合中国用户习惯的一组扩展功能，包括：节点的圆形图片化显示、数据驱动下节点大小和颜色呈现、关系连线的粗细和颜色设置，以及通用智能查询等功能。

本章最后一节还将介绍一个能够方便导入 Excel、MySQL 等数据的转换工具——导入精灵。

本章包括以下内容：

- Neo4j 简体中文版简介
- Neo4j 简体中文版的下载及安装、启动及浏览
- 如何将节点显示为图片
- 如何控制节点的尺寸和颜色
- 如何控制关系的粗细和颜色
- 如何使用智能查询
- 如何使用导入精灵

9.1 简介

由微云数聚公司开发并发布的 Neo4j 简体中文版是为中国用户量身定制的图数据库解决方案，除提供简体中文界面外，还提供了更好的可视化效果和更易用的查询功能。简体中文版对 Neo4j 扩展的功能如下：

（1）节点可以显示为图片，使显示效果更加生动。所支持的图片类型包括静态的 jpg 和动态的 gif 格式。下面两张图分别是 Neo4j 原来的效果和扩展后支持图片的效果，如图 9-1、图 9-2 所示。

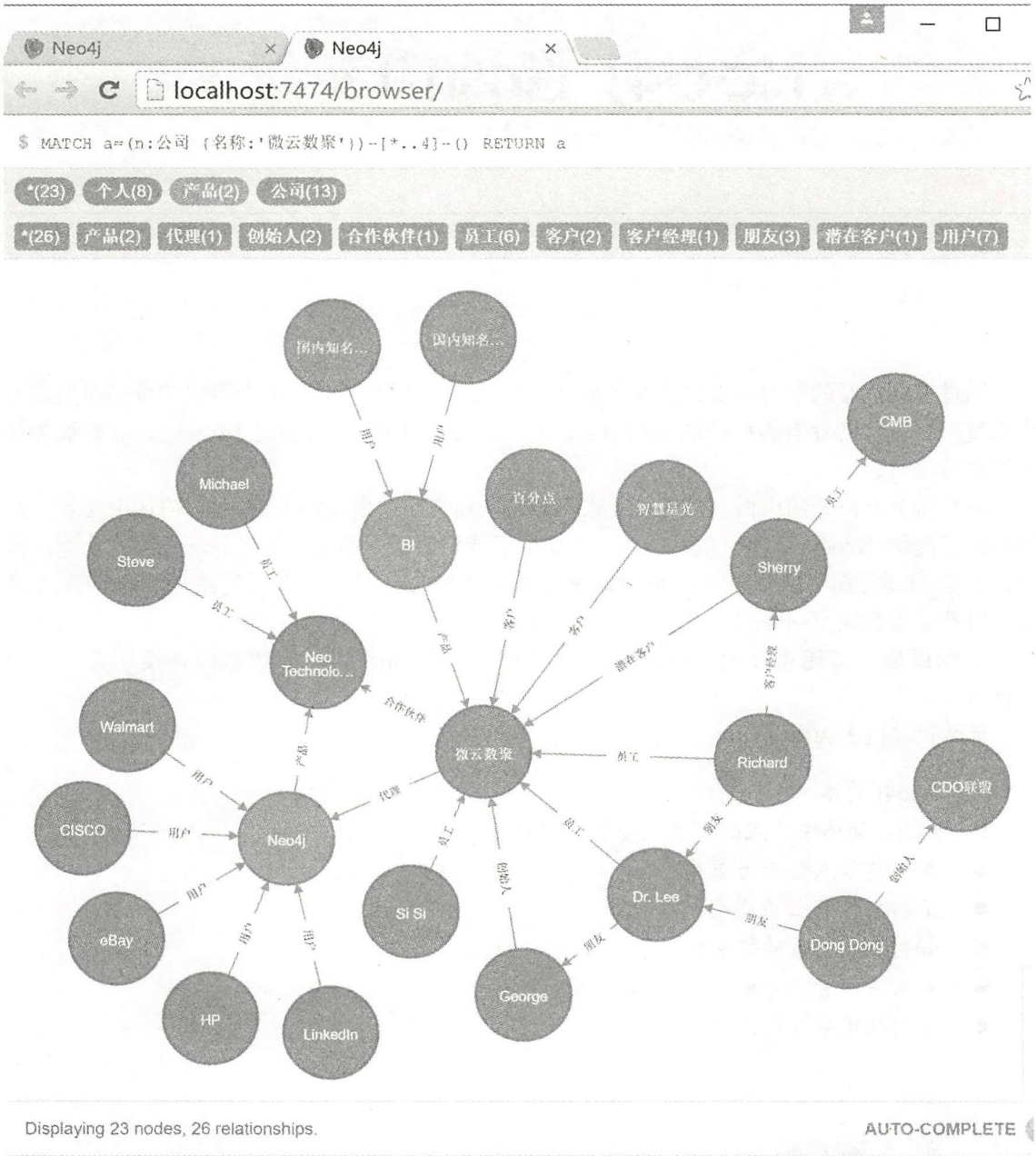


图 9-1　Neo4j 原来的效果

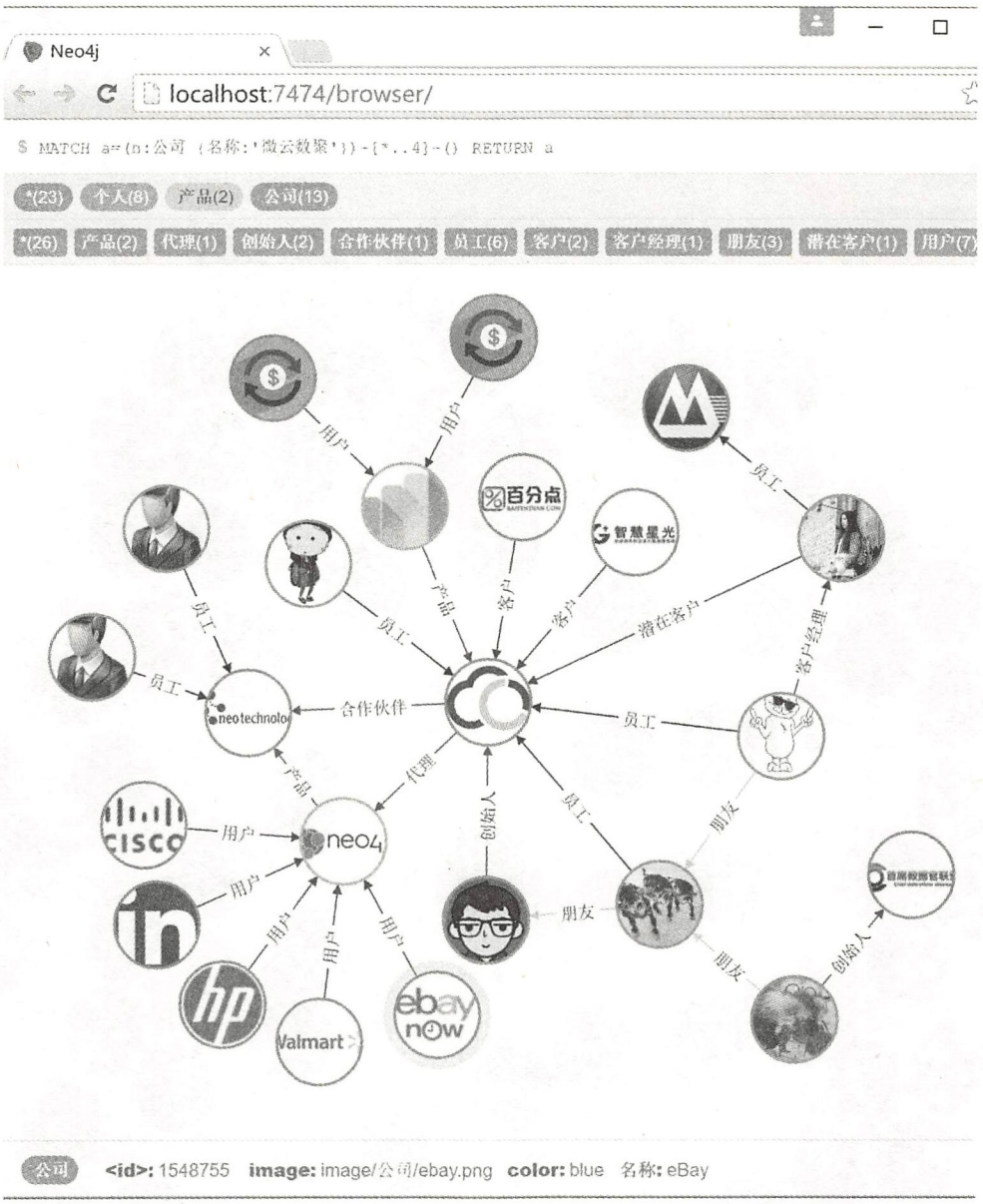


图 9-2　扩展后支持图片的效果

对比这两张图可以看到，扩展后比扩展前的效果要更加生动。

（2）节点的大小和颜色以及关系的粗细和颜色都可由数据驱动，使显示效果更加直观明了。下面两张图分别是 Neo4j 原来的效果和扩展后的效果，如图 9-3、图 9-4 所示。

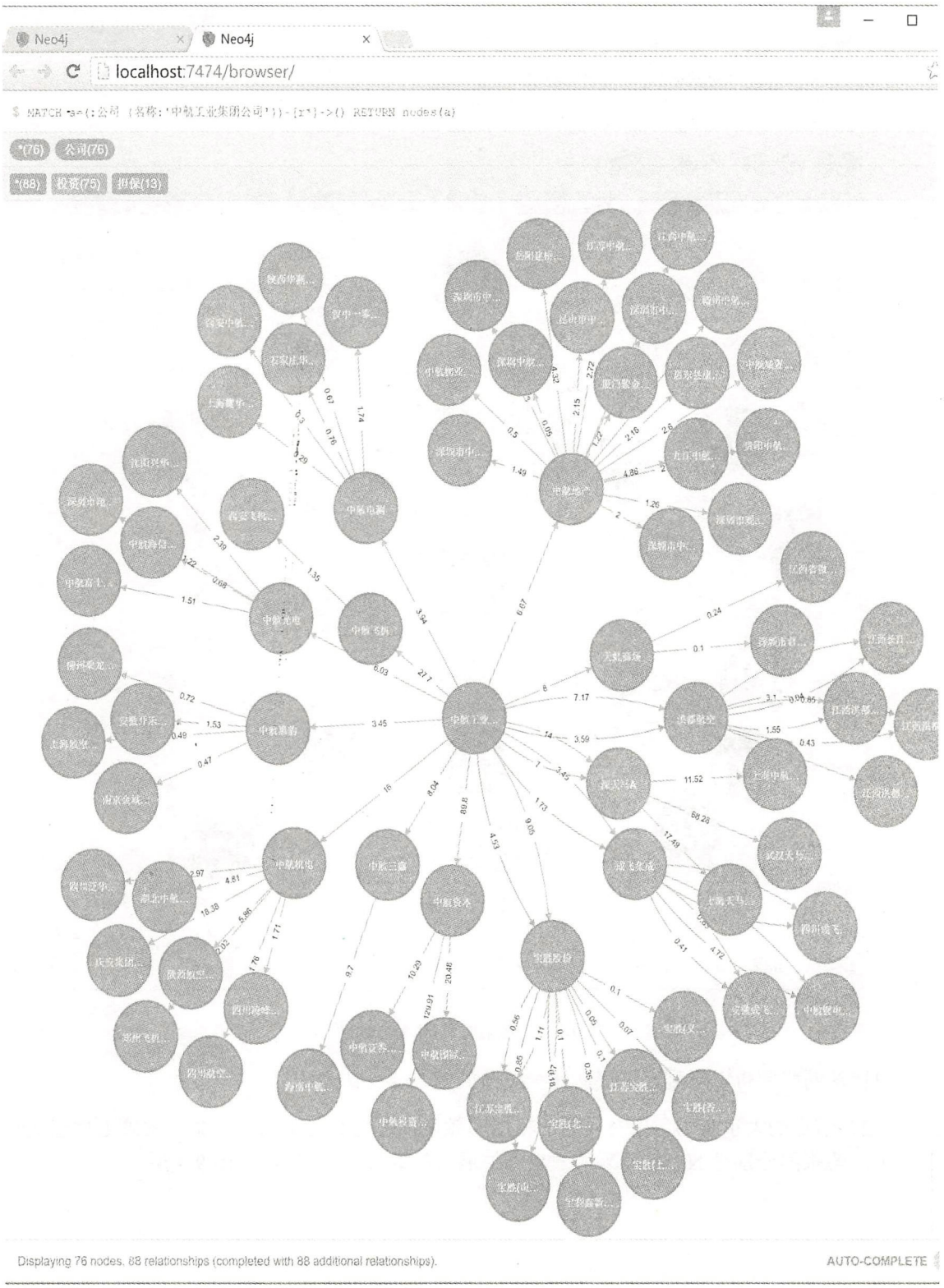


图 9-3　Neo4j 原来的效果

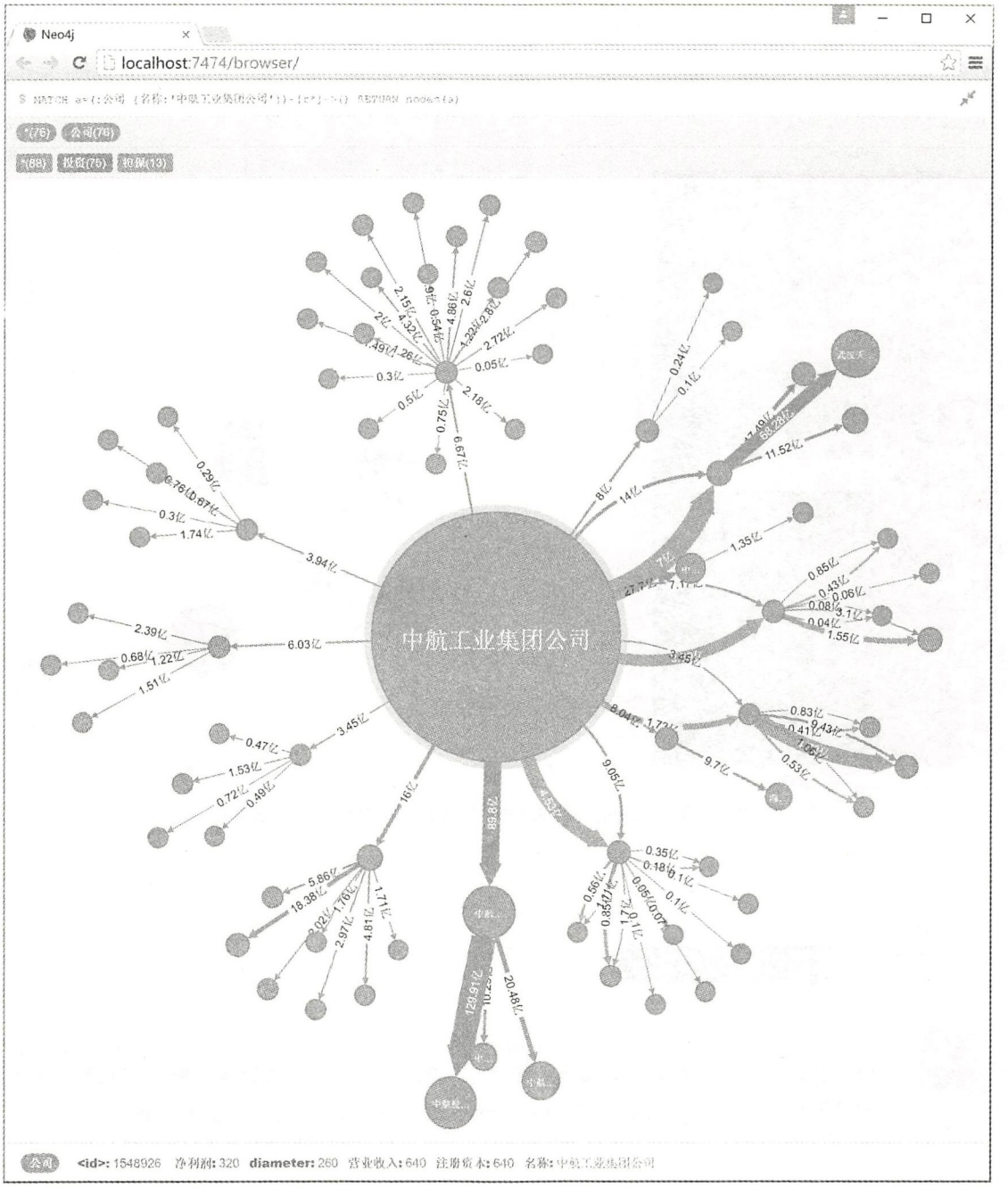


图 9-4　扩展后节点和关系由数据驱动的效果

后面这张图展示的是某集团公司的投资、担保关系图。节点的大小反映公司的注册资本的大小，红线表示投资关系，其粗细表示投资金额的大小。绿线表示担保关系，其粗细表示担保金额的大小。

我们可以很明显地看到，扩展后的效果更加直观。

（3）提供了智能查询功能，用户不需编写 Cypher 语句，也能像操作 Google 搜索一

样，方便地查询节点及节点之间的关系。这使得图数据库对最终用户更加友好，如图 9-5 所示。

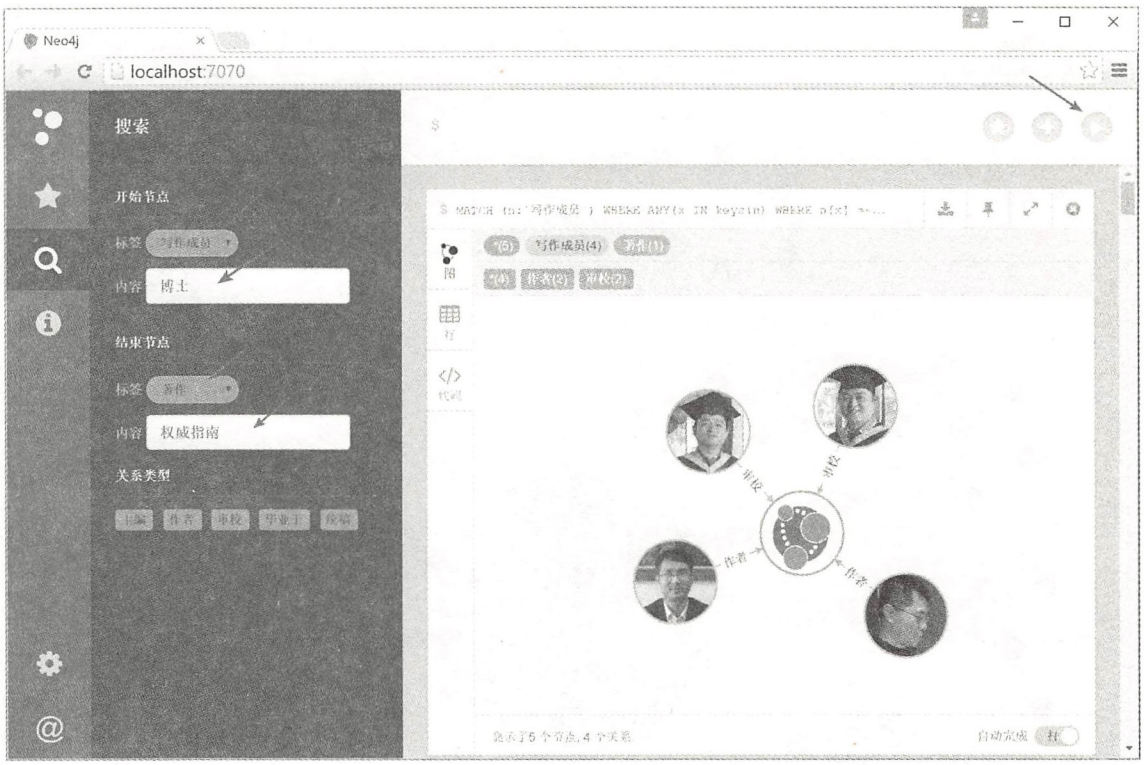


图 9-5　像 Google 一样方便地查询节点和节点之间的关系

9.2 下载及安装

提示

（1）目前仅为读者提供 Windows 版本；（2）如果之前安装过英文版，需到英文版的 bin 目录中执行 neo4j.bat stop 和 neo4j.bat uninstall-service，将原有服务卸载。

1. 下载

从下列地址可下载 Neo4j 所有版本，包括简体中文版（当前最新版本是 3.1.3）：

http://www.we-yun.com/index.php/blog/releases-56.html

打开的页面如图 9-6 所示。



图 9-6　下载 Neo4j 简体中文版

单击“neo4j community 3.1.3 windows 简体中文版 1.0.0 (zip)”并下载文件 neo4j-community-3.1.3-windows-chs-1.0.0.zip （建议用迅雷软件下载）。

2. 安装

安装非常简单，只需展开已下载的 neo4j-community-3.1.3-windows-chs-1.0.0.zip 文件（建议用 7-Zip 展开）。

9.3 启动及浏览

1. 启动

打开文件资源管理器，进入安装文件夹下的子文件夹 bin，右击 neo4j-start.cmd，选择“以管理员身份运行(A)”即可启动服务，如图 9-7 所示。同样，右击 neo4j-stop.cmd 则停止服务。

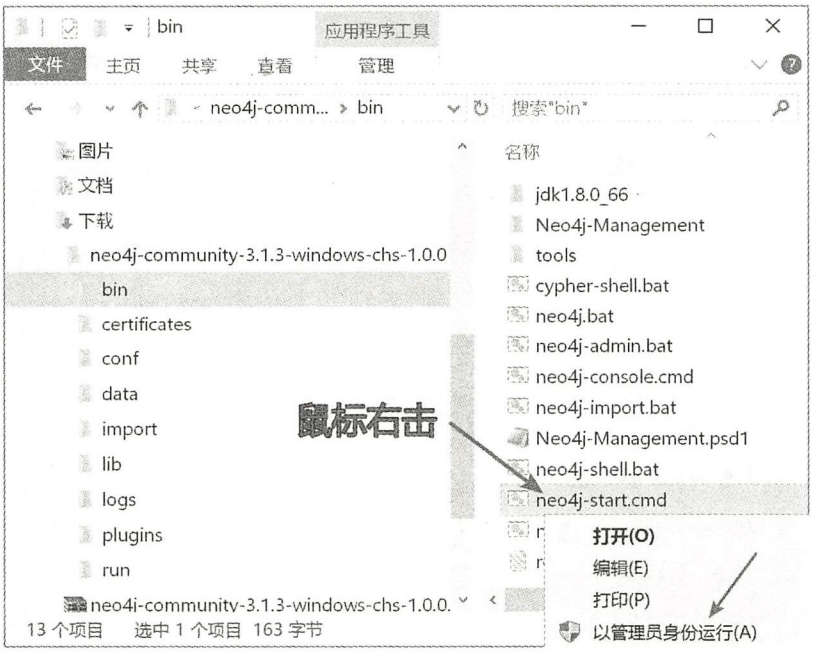


图 9-7　启动服务

2. 浏览

打开浏览器进入中文界面：http://localhost:7474/，如图 9-8 所示。



图 9-8　进入中文界面

9.4 指定节点图片

节点属性	image
属性值	图片的相对路径或绝对路径，如"image/公司/微云数聚.jpg"
功能	将节点显示为指定圆形图片

要将节点显示为图片，需要做两件事：

（1）将图片存放到服务器。本章所用图片均已事先准备好，并存放在下述文件夹：

C:\neo4j-chs\neo4j-web-cn\webapps\ROOT

（2）在节点中设置属性 image，用其指定图片的相对路径（也可以是绝对路径）。

例：创建带图片的节点。

```
CREATE
  (张老师:写作成员 {名称:"张老师", image:"image/作者/张老师.jpg"}),
  (苏博士:写作成员 {名称:"苏博士", image:"image/作者/苏博士.jpg"}),
  (庞砖家:写作成员 {名称:"庞砖家", image:"image/作者/庞砖家.jpg"}),
  (陈博士:写作成员 {名称:"陈博士", image:"image/作者/陈博士.jpg"}),
  (高博士:写作成员 {名称:"高博士", image:"image/作者/高博士.jpg"}),
  (李博士:写作成员 {名称:"李博士", image:"image/作者/李博士.jpg"}),
  (薛大侠:写作成员 {名称:"薛大侠", image:"image/作者/薛大侠.jpg"}),
  (赵大侠:写作成员 {名称:"赵大侠", image:"image/作者/赵大侠.jpg"}),
  (琴琴:写作成员 {名称:"琴琴", image:"image/作者/琴琴.jpg"}),
  (家辉:写作成员 {名称:"家辉", image:"image/作者/家辉.jpg"}),
  (国防科大:大学 {名称:"国防科大", image:"image/大学/国防科大.jpg"}),
  (枣庄学院:大学 {名称:"枣庄学院", image:"image/大学/枣庄学院.jpg"}),
  (中科院:大学 {名称:"中科院", image:"image/大学/中科院.jpg"}),
  (西安交大:大学 {名称:"西安交大", image:"image/大学/西安交大.jpg"}),
  (北京邮大:大学 {名称:"北京邮大", image:"image/大学/北京邮大.jpg"}),
  (北体大:大学 {名称:"北体大", image:"image/大学/北体大.jpg"}),
  (西南交大:大学 {名称:"西南交大", image:"image/大学/西南交大.jpg"}),
  (权威指南:著作 {名称:"Neo4j 权威指南", image:"image/著作/著作.jpg"}),
  (张老师)-[:毕业于]->(国防科大),
  (苏博士)-[:毕业于]->(国防科大),
  (庞砖家)-[:毕业于]->(枣庄学院),
  (陈博士)-[:毕业于]->(中科院),
  (高博士)-[:毕业于]->(中科院),
  (李博士)-[:毕业于]->(中科院),
  (薛大侠)-[:毕业于]->(西安交大),
  (赵大侠)-[:毕业于]->(北京邮大),
  (琴琴)-[:毕业于]->(北体大),
  (家辉)-[:毕业于]->(西南交大),
  (张老师)-[:主编]->(权威指南),
  (庞砖家)-[:作者]->(权威指南),
  (陈博士)-[:审校]->(权威指南),
  (高博士)-[:审校]->(权威指南),
  (李博士)-[:作者]->(权威指南),
```

```
(薛大侠)-[:审校]->(权威指南),
(苏博士)-[:作者]->(权威指南),
(赵大侠)-[:作者]->(权威指南),
(琴琴)-[:统稿]->(权威指南),
(家辉)-[:作者]->(权威指南),
(家辉)-[:审校]->(权威指南)
```

在界面输入上述语句，如图 9-9 所示。

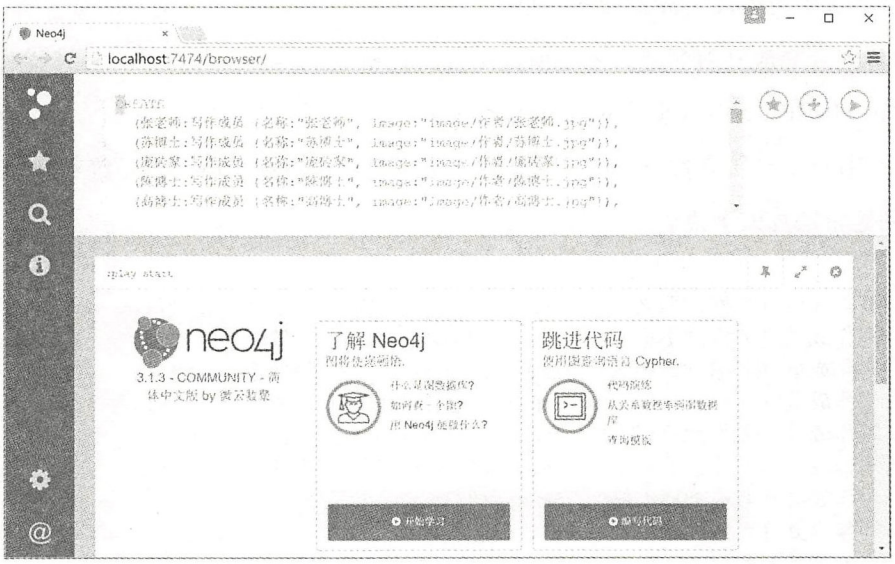


图 9-9　创建带图片的节点

查看结果（查询 1 度关系），如图 9-10 所示。

```
MATCH m=(:著作 {名称:"Neo4j 权威指南"})--() RETURN m
```

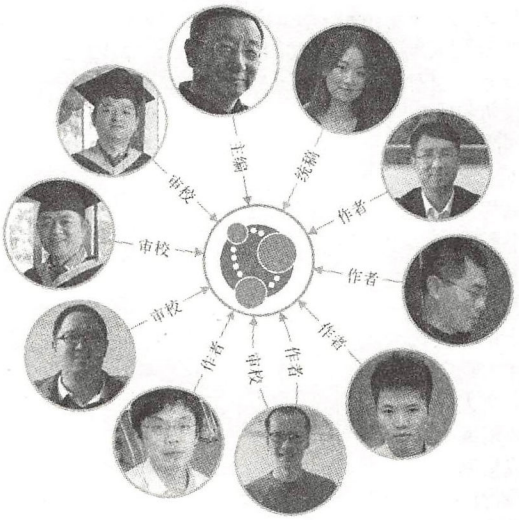


图 9-10　显示带图片的节点（1 度）

查看结果（查询 2 度关系），如图 9-11 所示。

```
MATCH m=(:著作 {名称:"Neo4j 权威指南"})-[*..2]-() RETURN m
```

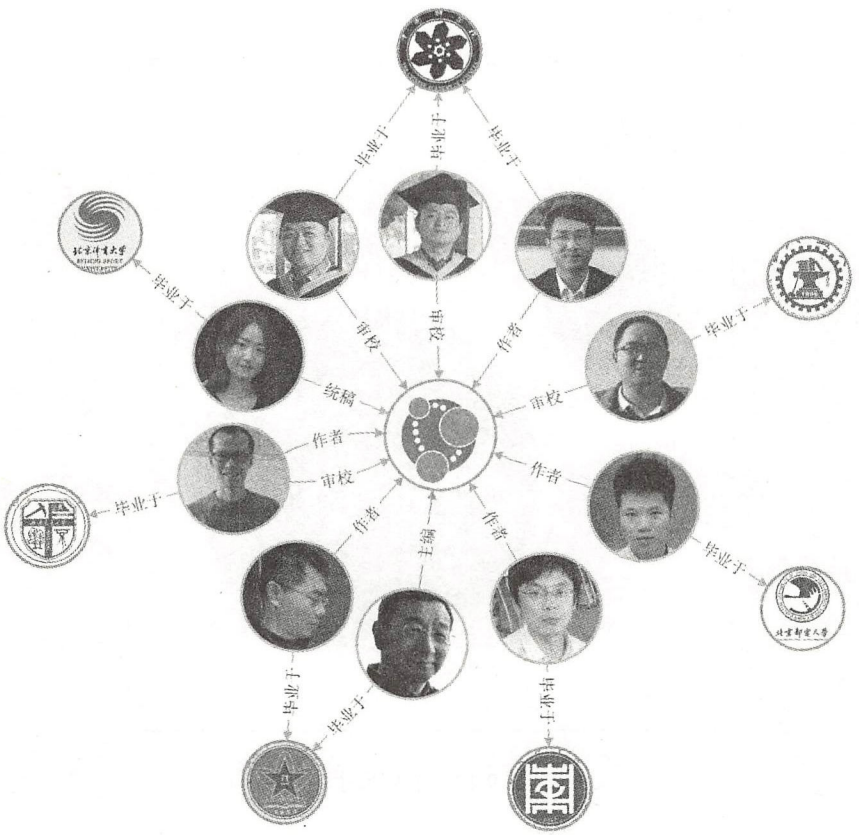


图 9-11　显示带图片的节点（2 度）

9.5 指定节点尺寸

节点属性	diameter
属性值	节点的直径，如 21.09，53.68，260.00
功能	指定节点的显示尺寸

例：指定节点尺寸。

```
CREATE (n:Company {名称:'中航工业集团公司'})
SET n.注册资本=640,n.营业收入=640,n.净利润=320,n.diameter=260
```

```
CREATE (n:Company {名称:'宝胜股份'})
SET n.注册资本=9.05,n.营业收入=9.05,n.净利润=4.53,n.diameter=100
```

查看结果，如图 9-12 所示。

```
MATCH (n:Company) RETURN n
```



图 9-12　节点尺寸

9.6 指定节点颜色

节点属性	color
属性值	节点的颜色，如 blue，red，yellow 等
功能	指定节点的显示颜色

例：指定节点颜色。

```
MATCH (n:Company {名称:'中航工业集团公司'})
SET n.color='red'
```

```
MATCH (n:Company {名称:'宝胜股份'})
SET n.color='blue'
```

查看结果，如图 9-13 所示。

```
MATCH (n:Company) RETURN n
```

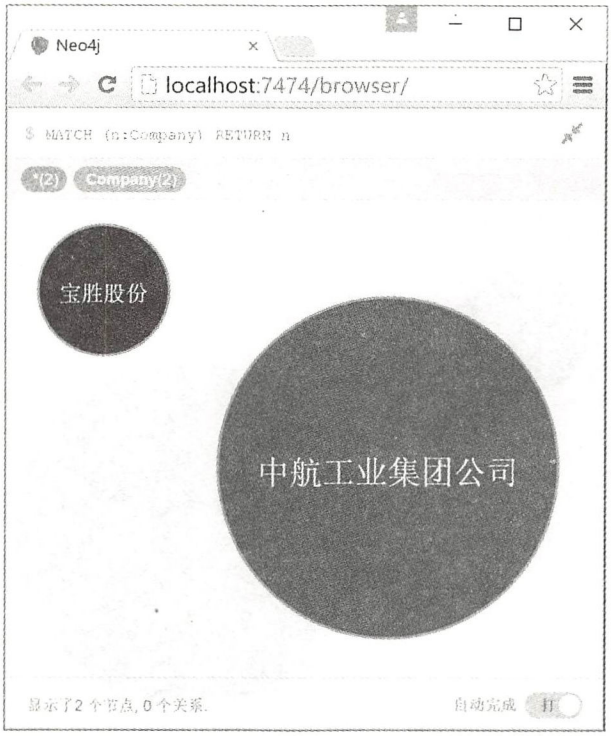


图 9-13　节点颜色

9.7 指定关系粗细

节点属性	shaftWidth
属性值	连线的粗细，如 1.33, 14.13, 25.98
功能	指定关系连线的显示粗细

例：创建关系并指定关系的粗细。

```
MATCH (n:Company {名称:'中航工业集团公司'}), (m:Company {名称:'宝胜股份'})
MERGE (n)-[r:投资]->(m)
SET r.持股比例=100,r.投资金额=9.05,r.shaftWidth=5
```

```
MATCH (n:Company {名称:'中航工业集团公司'}), (m:Company {名称:'宝胜股份'})
MERGE (n)-[r:担保]->(m)
SET r.担保金额=4.53,r.shaftWidth=15.16
```

查看结果，如图 9-14 所示。

```
MATCH (n:Company) RETURN n
```



图 9-14　关系粗细

9.8 指定关系颜色

节点属性	color
属性值	连线的颜色，如 blue, red, yellow 等
功能	指定关系连线的显示颜色

例：创建关系并指定关系的颜色。

```
MATCH (n:Company {名称:'中航工业集团公司'}), (m:Company {名称:'宝胜股份'})
MERGE (n)-[r:投资]->(m)
SET r.color='red'
```

```
MATCH (n:Company {名称:'中航工业集团公司'}), (m:Company {名称:'宝胜股份'})
MERGE (n)-[r:担保]->(m)
SET r.color='green'
```

查看结果，如图 9-15 所示。

```
MATCH (n:Company) RETURN n
```

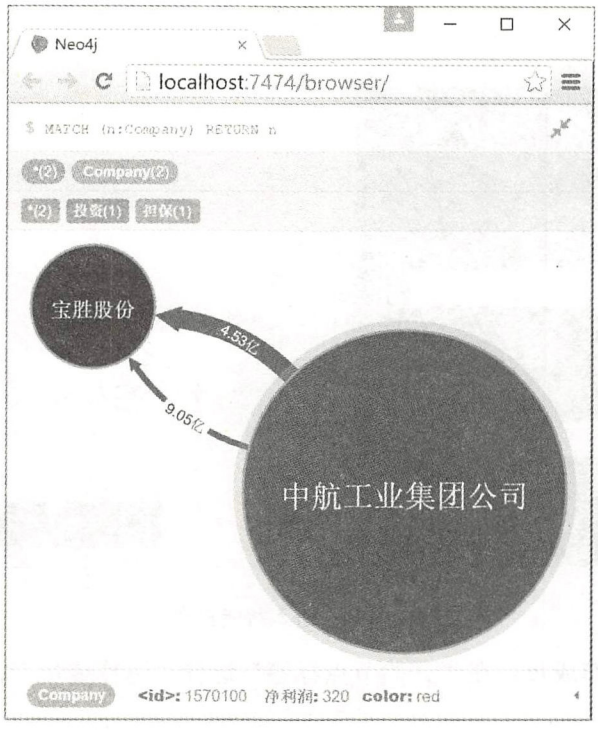


图 9-15　关系颜色

9.9 智能查询

智能查询功能允许用户不需编程就能进行各种常用查询。可以通过输入一组条件（开始节点标签及内容）查询节点，也可通过输入两组条件（开始节点标签及内容，结束节点标签及内容）查询节点与节点之间的关系：

- “开始节点标签”：选择开始节点标签。
- “开始节点内容”：输入开始节点内容。
- “结束节点标签”：选择结束节点标签。
- “结束节点内容”：输入结束节点内容。
- “操作”：执行查询。

如果只选择“开始节点标签”，则查询出指定标签的所有节点。

如果还输入了“开始节点内容”，则查询出指定标签的所有包含该内容的所有节点，注意，此处是模糊查询。

如果指定了“结束节点标签”和输入了“结束节点内容”，则会查询出所有“开始节点”和“结束节点”以及这些节点之间的关系。

下面举例说明其用法。首先，在 Neo4j 浏览器界面单击左侧的查询按钮，如图 9-16 所示。



图 9-16　进入智能查询

例：查询所有写作成员。在“开始节点标签”选择“写作成员”，单击操作按钮，如图 9-17 所示。

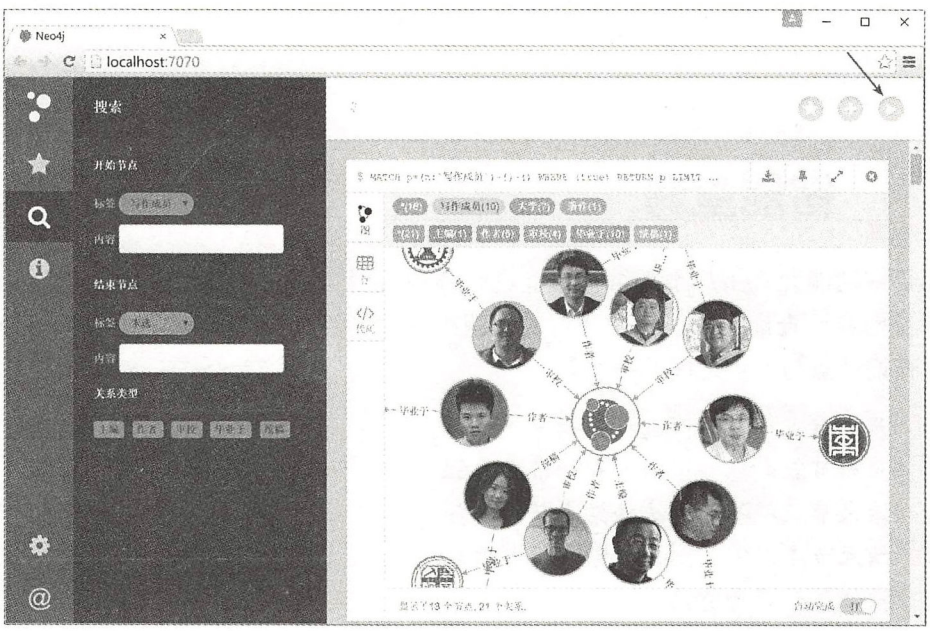


图 9-17　查询所有写作成员

例：查询所有名称中包含博士的写作成员：在“开始节点标签”选择“写作成员”，在“开始节点内容”输入“博士”，单击操作按钮，如图 9-18 所示。

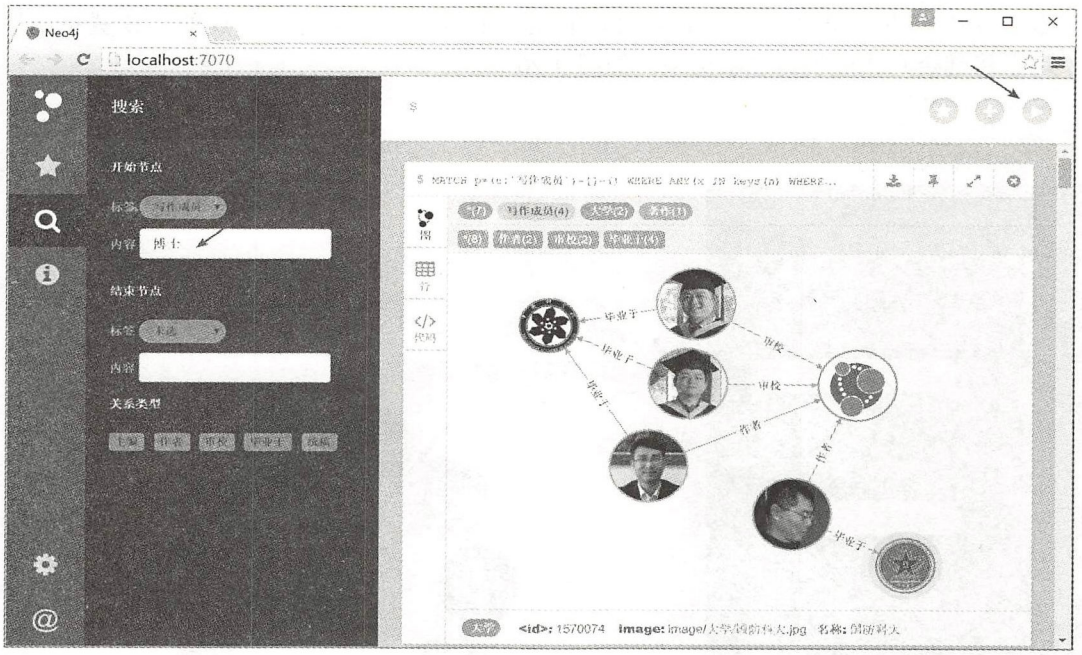


图 9-18　查询所有写作成员中的博士

例：查询博士写的包含“权威指南”的著作：在“开始节点标签”选择“写作成员”，在“开始节点内容”输入“博士”，在“结束节点标签”选择“著作”，在“结束节点内容”输入“权威指南”，单击操作按钮，如图 9-19 所示。

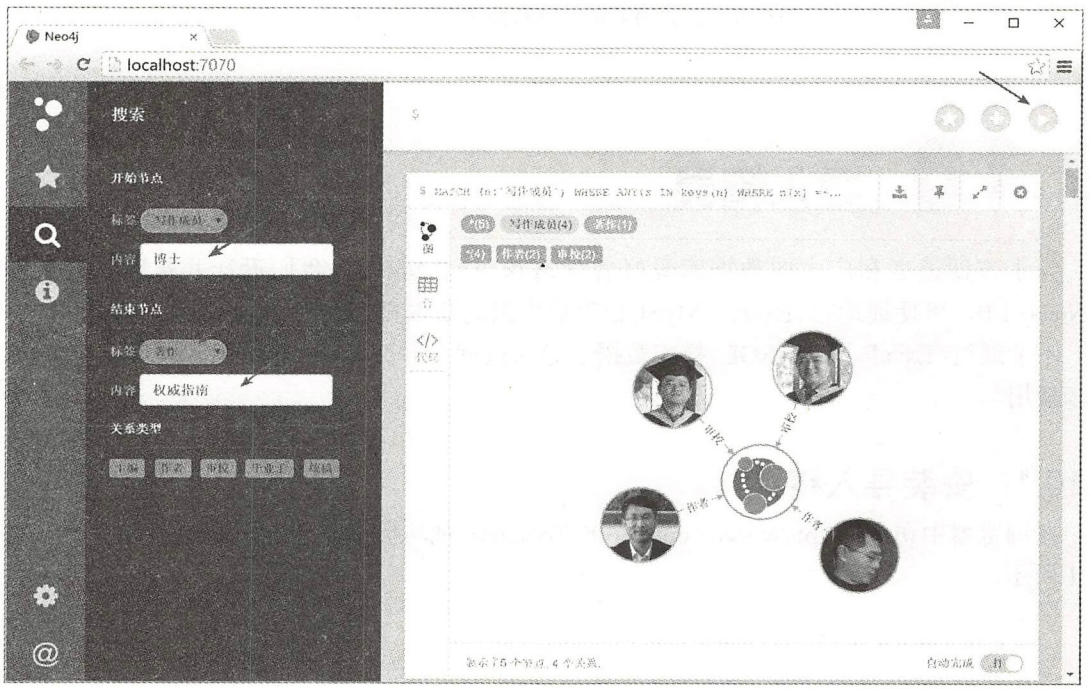


图 9-19　查询写《权威指南》写作成员中的博士

双击图中节点，可以查到该节点的其他相关节点。比如双击正中间的“权威指南”节点，可以将其余写作成员也显示出来，如图 9-20 所示。

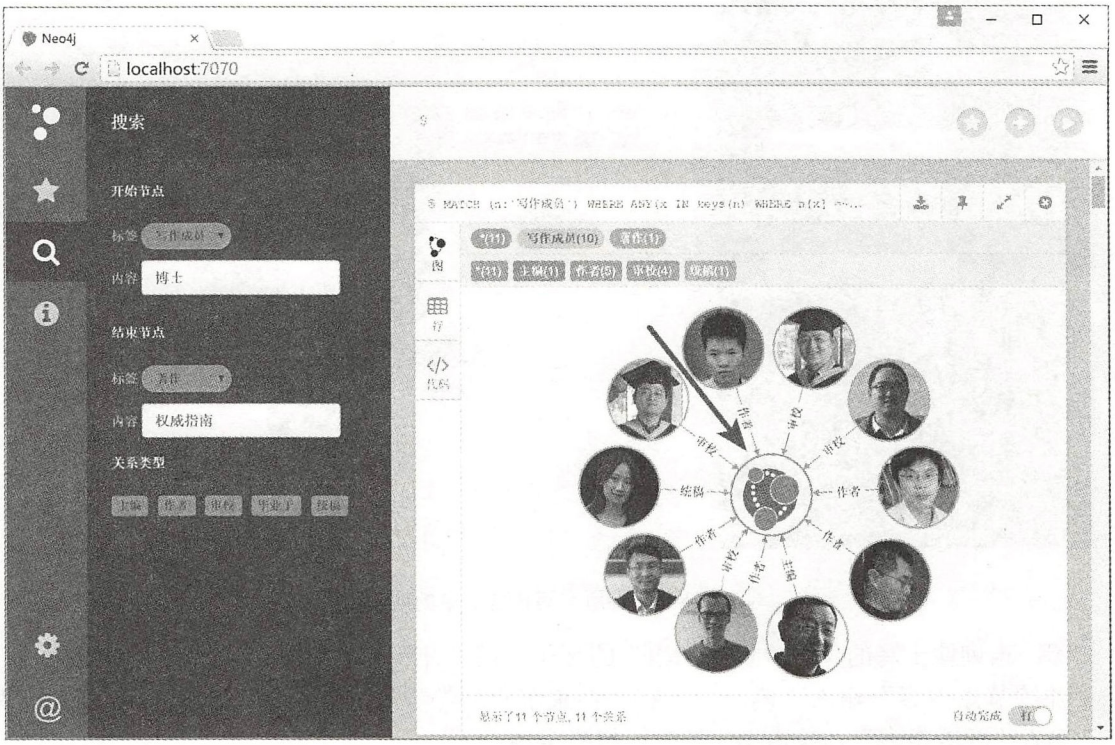


图 9-20　双击查看所有《权威指南》的写作成员

9.10 导入精灵

为了方便读者和广大图数据库爱好者学习和研究 Neo4j，我们设计并实现了导入精灵 ToNeo4j 1.0，可便捷地将 Excel、MySQL 等数据源的数据直接导入 Neo4j 数据库。

本节通过 Excel 和 MySQL 样本数据的导入过程，详细介绍如何使用该工具，包括其安装和使用。

9.10.1 安装导入精灵

在浏览器中访问 http://we-yun.com:8000/ToNeo4j 便可完成导入精灵的下载和安装，如图 9-21 所示。

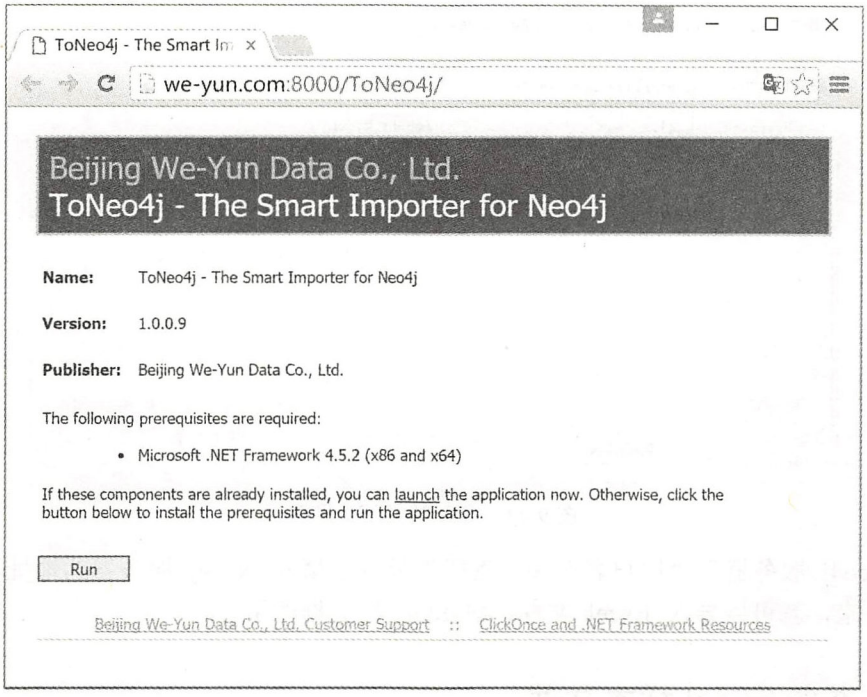


图 9-21　导入精灵安装页面

9.10.2　启动导入精灵

安装完成后，将自动启动导入精灵界面。导入精灵的界面是系统语言敏感的，即根据系统显示语言不同，其界面会自动选用相应语言。下面是导入精灵的简体中文界面和英文界面，如图 9-22、图 9-23 所示。

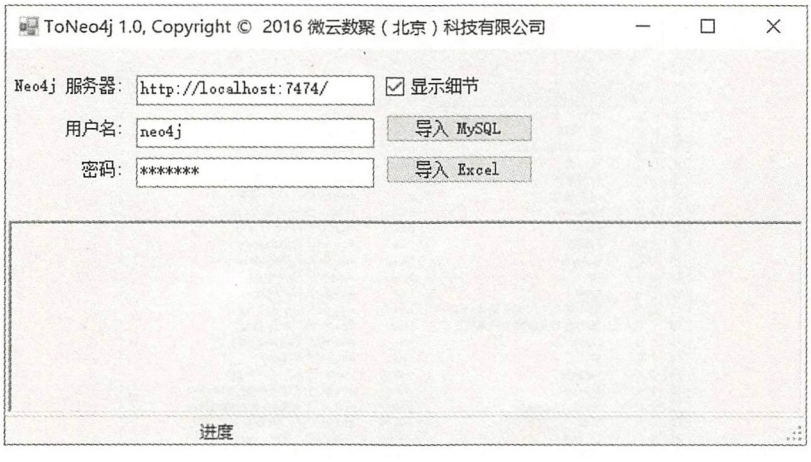


图 9-22　导入精灵简体中文界面

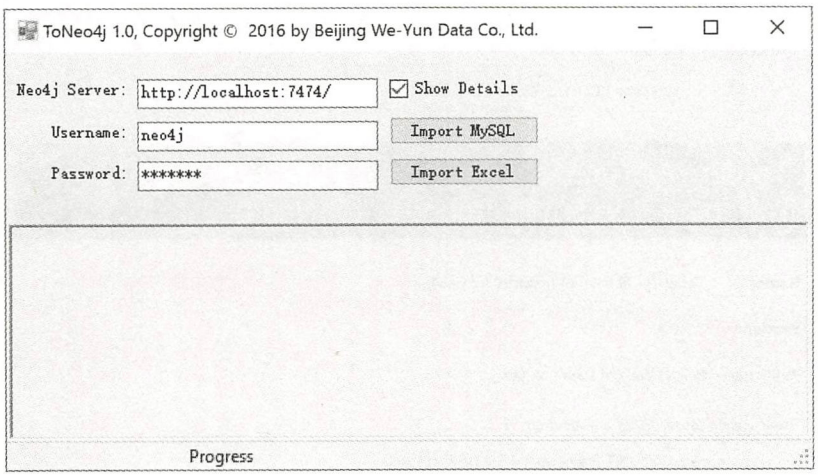


图 9-23　导入精灵英文界面

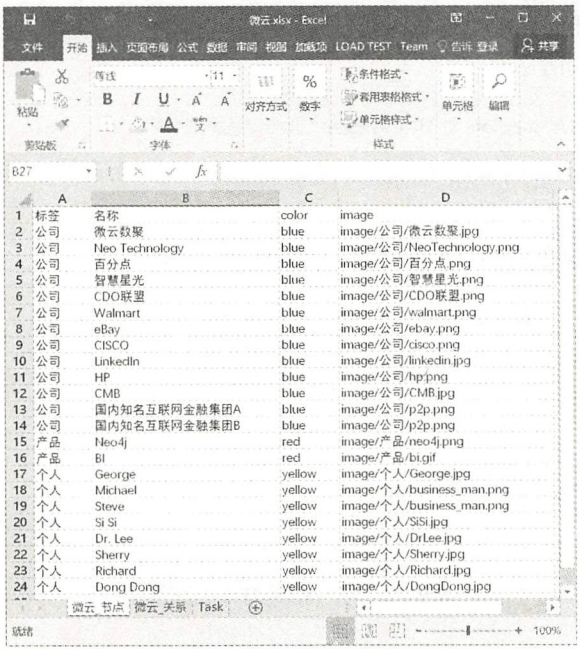
在“Neo4j 服务器”“用户名”和“密码”处分别填入 Neo4j 服务器的地址、用户名和密码。接下来，就可以导入 Excel 文件、MySQL 等数据库了。

9.10.3　准备 Excel 样本文件

先下载演示用的 Excel 文件。下载地址是：

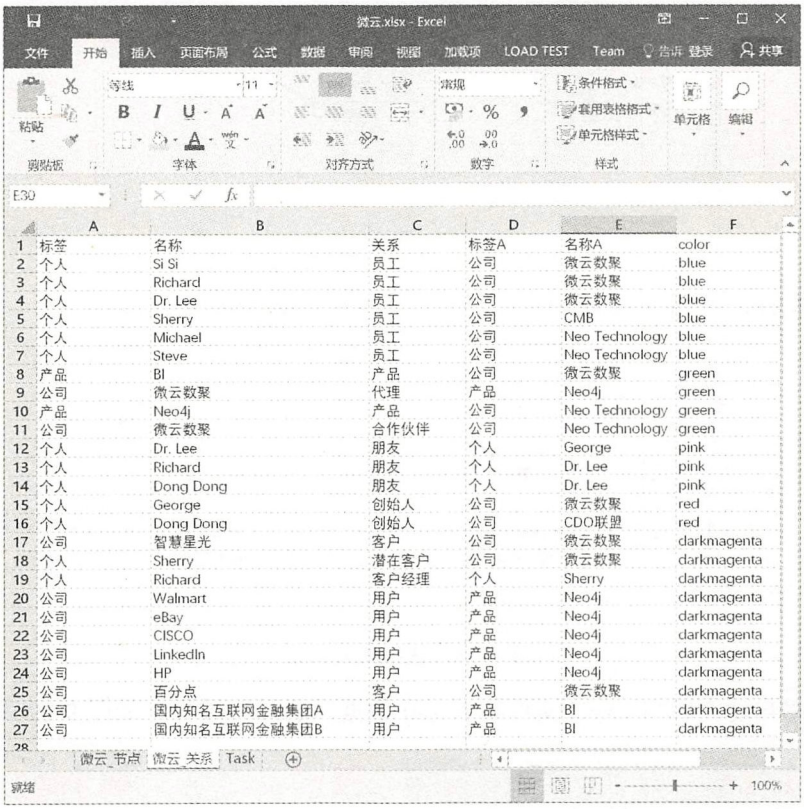
http://we-yun.com/neo4jguide/微云.xlsx

该文件包含三个表：“微云_节点”“微云_关系”和“微云_task”，如图 9-24~图 9-26 所示。



	A	B	C	D
1	标签	名称	color	image
2	公司	微云数聚	blue	image/公司/微云数聚.jpg
3	公司	Neo Technology	blue	image/公司/NeoTechnology.png
4	公司	百分点	blue	image/公司/百分点.png
5	公司	智慧星光	blue	image/公司/智慧星光.png
6	公司	CDO联盟	blue	image/公司/CDO联盟.png
7	公司	Walmart	blue	image/公司/walmart.png
8	公司	eBay	blue	image/公司/ebay.png
9	公司	CISCO	blue	image/公司/cisco.png
10	公司	LinkedIn	blue	image/公司/linkedin.jpg
11	公司	HP	blue	image/公司/hp.png
12	公司	CMB	blue	image/公司/CMB.jpg
13	公司	国内知名互联网金融集团A	blue	image/公司/p2p.png
14	公司	国内知名互联网金融集团B	blue	image/公司/p2p.png
15	产品	Neo4j	red	image/产品/neo4j.png
16	产品	BI	red	image/产品/bi.gif
17	个人	George	yellow	image/个人/George.jpg
18	个人	Michael	yellow	image/个人/business_man.png
19	个人	Steve	yellow	image/个人/business_man.png
20	个人	Si Si	yellow	image/个人/SiSi.jpg
21	个人	Dr. Lee	yellow	image/个人/DrLee.jpg
22	个人	Sherry	yellow	image/个人/Sherry.jpg
23	个人	Richard	yellow	image/个人/Richard.jpg
24	个人	Dong Dong	yellow	image/个人/DongDong.jpg

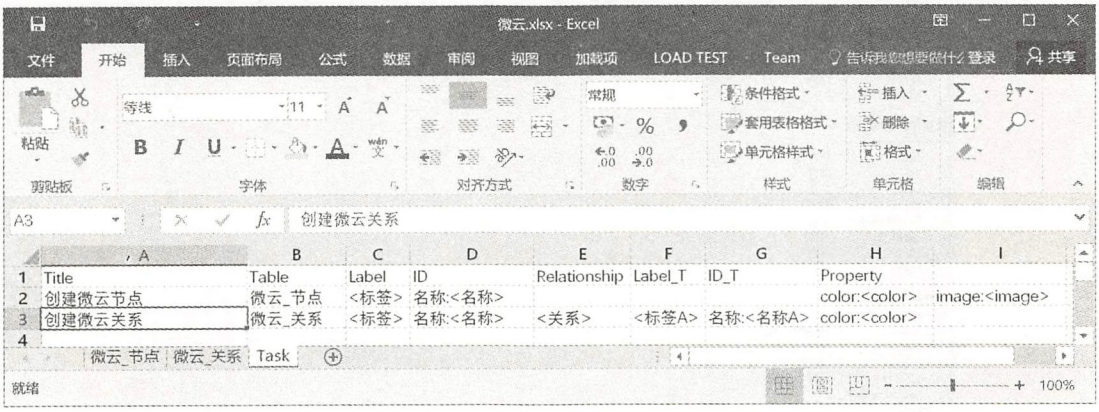
图 9-24　“微云_节点”表



微云.xlsx - Excel

	A	B	C	D	E	F
1	标签	名称	关系	标签A	名称A	color
2	个人	Si Si	员工	公司	微云数聚	blue
3	个人	Richard	员工	公司	微云数聚	blue
4	个人	Dr. Lee	员工	公司	微云数聚	blue
5	个人	Sherry	员工	公司	CMB	blue
6	个人	Michael	员工	公司	Neo Technology	blue
7	个人	Steve	员工	公司	Neo Technology	blue
8	产品	BI	产品	公司	微云数聚	green
9	公司	微云数聚	代理	产品	Neo4j	green
10	产品	Neo4j	产品	公司	Neo Technology	green
11	公司	微云数聚	合作伙伴	公司	Neo Technology	green
12	个人	Dr. Lee	朋友	个人	George	pink
13	个人	Richard	朋友	个人	Dr. Lee	pink
14	个人	Dong Dong	朋友	个人	Dr. Lee	pink
15	个人	George	创始人	公司	微云数聚	red
16	个人	Dong Dong	创始人	公司	CDO联盟	red
17	公司	智慧星光	客户	公司	微云数聚	darkmagenta
18	个人	Sherry	潜在客户	公司	微云数聚	darkmagenta
19	个人	Richard	客户经理	个人	Sherry	darkmagenta
20	公司	Walmart	用户	产品	Neo4j	darkmagenta
21	公司	eBay	用户	产品	Neo4j	darkmagenta
22	公司	CISCO	用户	产品	Neo4j	darkmagenta
23	公司	LinkedIn	用户	产品	Neo4j	darkmagenta
24	公司	HP	用户	产品	Neo4j	darkmagenta
25	公司	百分点	客户	公司	微云数聚	darkmagenta
26	公司	国内知名互联网金融集团A	用户	产品	BI	darkmagenta
27	公司	国内知名互联网金融集团B	用户	产品	BI	darkmagenta

图 9-25　“微云_关系”表



微云.xlsx - Excel

	A	B	C	D	E	F	G	H	I
1	Title	Table	Label	ID	Relationship	Label_T	ID_T	Property	
2	创建微云节点	微云_节点	<标签>	名称:<名称>				color:<color>	image:<image>
3	创建微云关系	微云_关系	<标签>	名称:<名称>	<关系>	<标签A>	名称:<名称A>	color:<color>	
4									

图 9-26　“微云_task”表

9.10.4　导入 Excel 样本文件

在界面单击“导入 Excel”按钮，开始导入 Excel 文件中的数据，如图 9-27 所示。如果不想在导入数据时显示细节，可以先取消“显示细节”选择框的勾选。

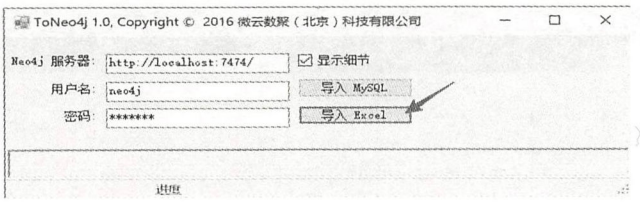


图 9-27　导入 Excel

选择导入文件，如图 9-28 所示。

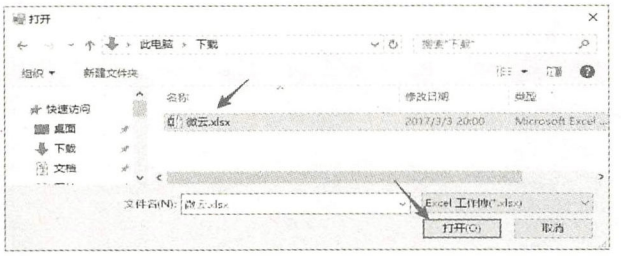


图 9-28　选择微云.xlsx

导入精灵将自动完成将 Excel 文件往 Neo4j 数据库的导入过程，如图 9-29 所示。

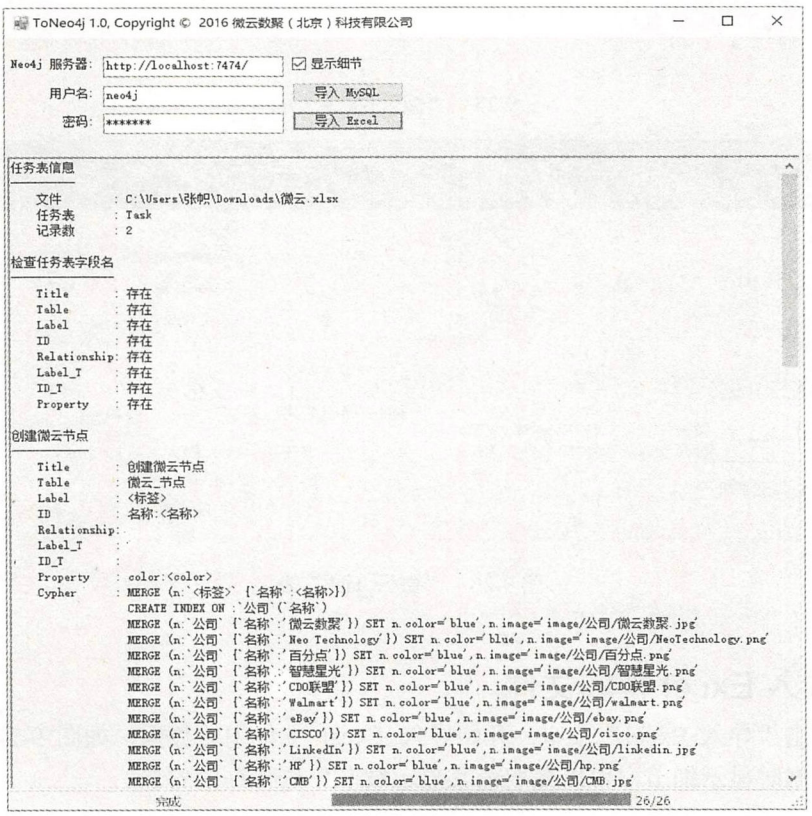


图 9-29　Excel 导入完成



导入后的效果请参见“9.10.7 查看导入结果”小节。

9.10.5　准备 MySQL 样本数据库

我们首先要准备好演示用的 MySQL 数据库，本书中采用的是 Navicat for MySQL 10.0.11。

（1）下载演示用的 MySQL 数据库备份文件。下载地址是：

http://we-yun.com/neo4jguide/微云.sql

（2）在 MySQL 中创建库 neo4jguide，“字符集”选择 utf8 – UTF-8 Unicode，“排序规则”选择 utf8_general_ci，如图 9-30 所示。

（3）在数据库 neo4jguide 上右击，选择“运行 SQL 文件”命令，开始导入 MySQL 数据库中的数据，如图 9-31 所示。

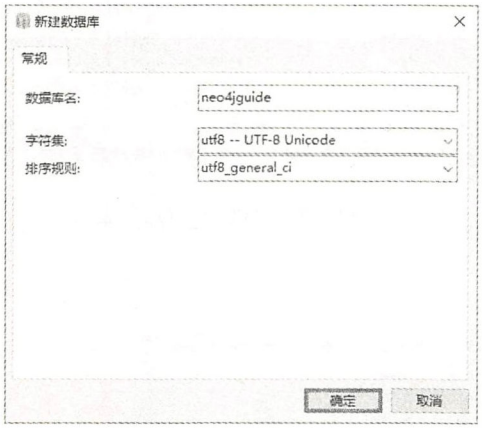


图 9-30　创建 neo4jguide 数据库



图 9-31　运行 SQL 文件

选择并打开“微云.sql”文件，如图 9-32 所示。

“编码”选择 65001（UTF-8）并单击“开始”按钮，如图 9-33 所示。



图 9-32　选择微云.sql 文件



图 9-33　开始导入 MySQL

完成导入，如图 9-34~图 9-36 所示。

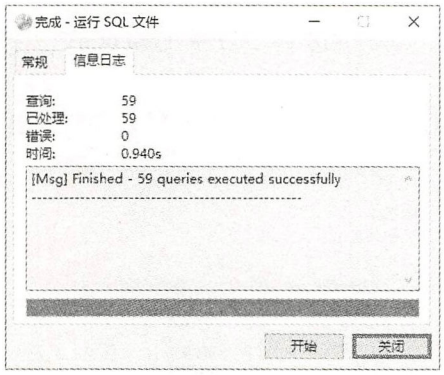


图 9-34 MySQL 导入完成

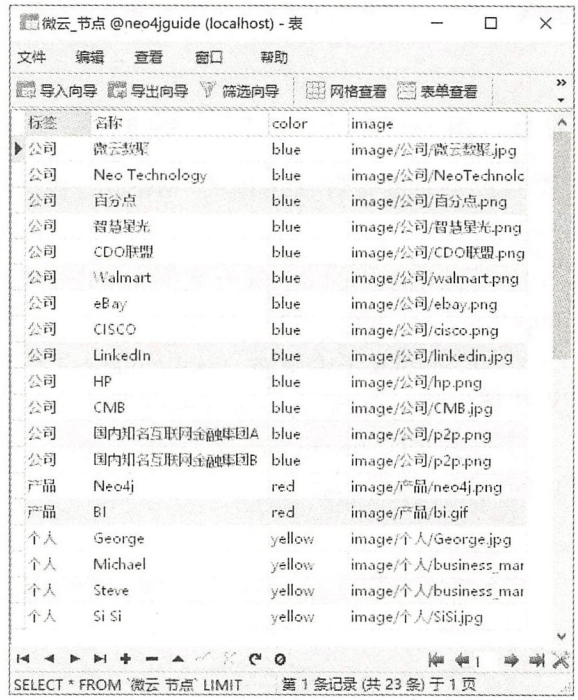


图 9-35 “微云_节点”表



图 9-36 “微云_关系”表

导入后，库 neo4jguide 中包含三个表格：“微云_节点”、“微云_关系”（如图 9-37

所示）、“微云_task”。



图 9-37　“微云_task”表

9.10.6　导入 MySQL 样本数据库

准备好演示用的 MySQL 数据库后，就可以用导入精灵导入 MySQL 数据库了：

（1）在导入精灵界面单击“导入 MySQL”按钮，如图 9-38 所示。如果不想在导入数据时显示导入细节，可以先取消 “显示细节”选择框的勾选。

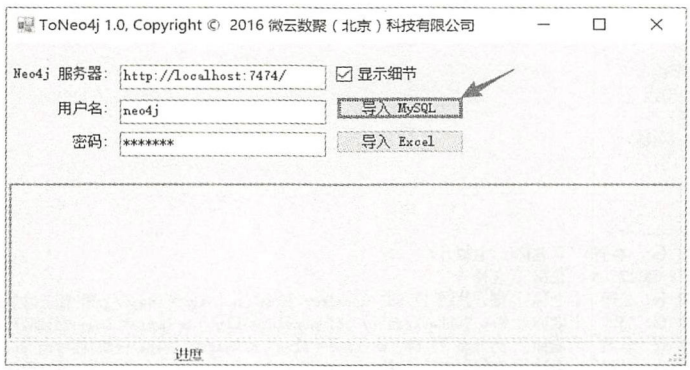


图 9-38　导入 MySQL

（2）输入服务器地址、端口、用户名、密码、数据库和表，单击“确认”按钮开始导入，如图 9-39 所示。

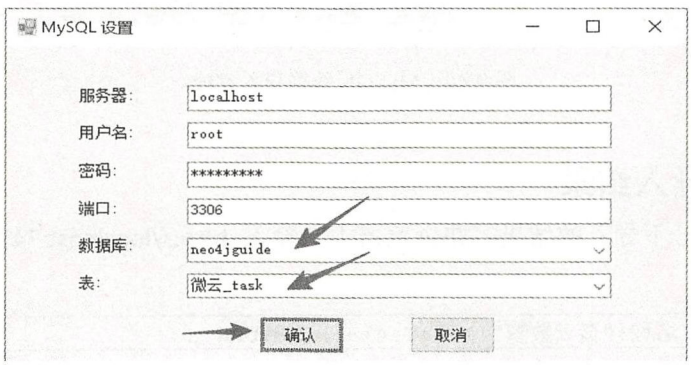


图 9-39　设置导入参数

（3）导入完成，如图 9-40 所示。

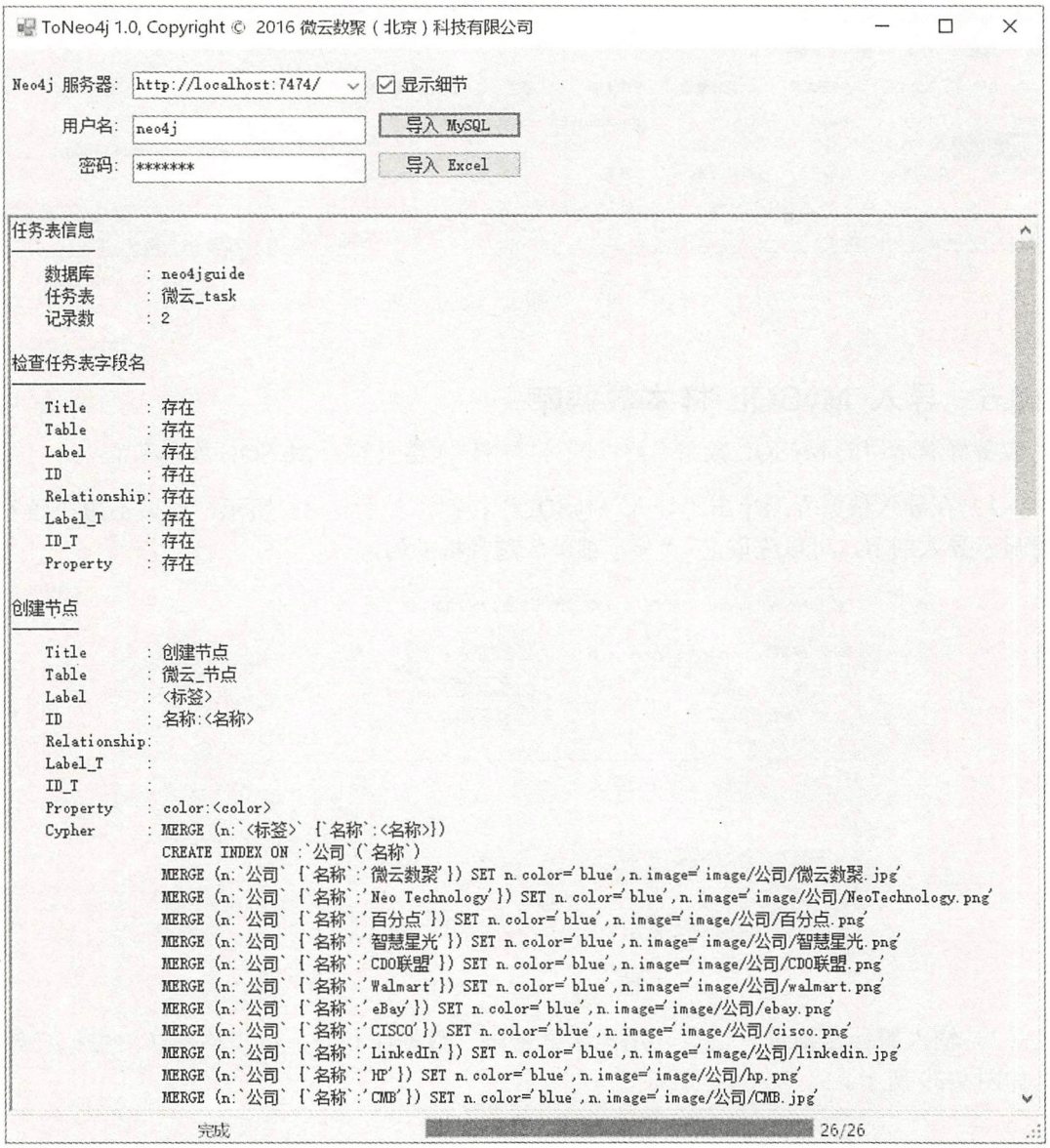


图 9-40　MySQL 数据导入完成

9.10.7　查看导入结果

现在可以查看一下导入的结果。在浏览器中，输入 http://localhost:7474，并执行如下命令：

```
MATCH a=(n:公司 {名称:"微云数聚"})-[*..4]-() RETURN a
```

结果如图 9-41 所示。

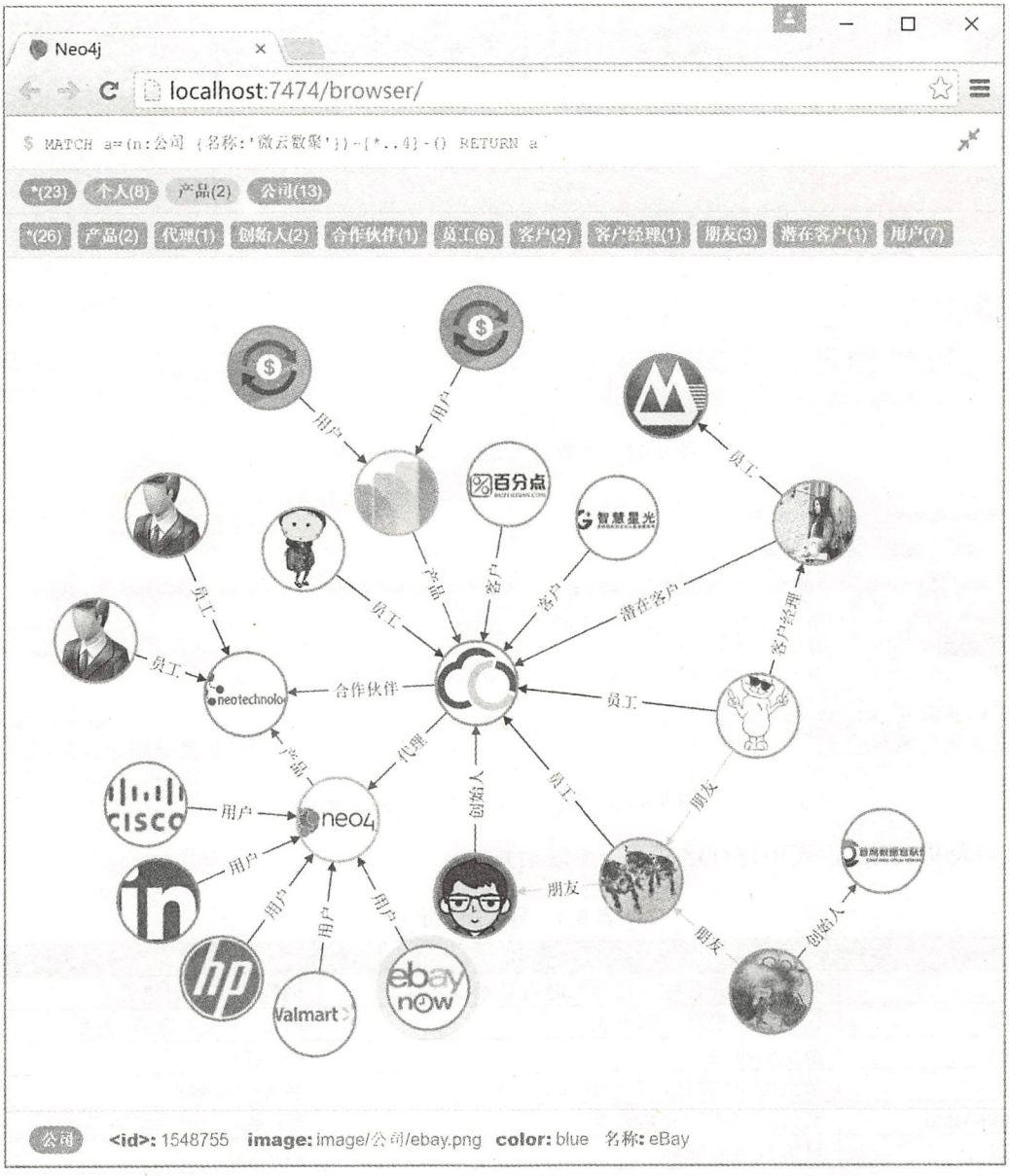


图 9-41　导入结果

9.10.8　导入精灵映射机制

大家知道，Excel 和 MySQL 都是表格形式的数据，而 Neo4j 是图数据库。那么，导入精灵是如何将表格型数据转换成图数据库结构的呢？这是靠导入精灵映射机制实现的。

导入精灵的映射规则定义在 task 表格中。Excel 中的 Task 表与 MySQL 中的 Task 表中定义的映射规则是一致的，如图 9-42 和图 9-43 所示。



图 9-42 “微云_task”表 (Excel)



图 9-43 “微云_task”表 (MySQL)

如表 9-1 所示，表中每行定义了一个映射规则。

表 9-1 导入规则定义

列名	含义	举例
Title	映射规则的名称，在导入时将显示该名称	创建节点、创建关系
Table	指明数据来源于哪个表	微云_节点、微云_关系
Label	节点的标签	<标签>
ID	节点的 ID 属性，其值在系统中唯一	名称:<名称>
Relationship	到目标节点的关系类型	<关系>
Label_T	目标节点的标签	<标签 A>
ID_T	目标节点的 ID 属性，其值在系统中唯一	名称:<名称>
Property	指定节点或关系的属性	color:<color>
Property 后面的列	指定其余属性	image:<image>

数据中以<>括起来的内容，表示来源于 Table 指定的表格。现在解释一下两行规则的具体含义：第一行定义“创建节点”规则，数据来源于“微云_节点”表，该规则将遍历“微云_节点”表的每一行，针对每一行创建一个节点。每个节点的标签名称来源于<标签>列，ID 属性为“名称”，其值来源于<名称>列，属性 color 的值来源于<color>列。

即针对“微云_节点”表的每一行，将创建形如下面的节点：

(:<标签> {名称:<名称>, color:<color>,image:<image>})

其中<标签>、<名称>、<color>、<image>的值取自于“微云_节点”表中每一行列名为标

签、名称、color、image 的值。其相应的映射关系如图 9-44 所示。

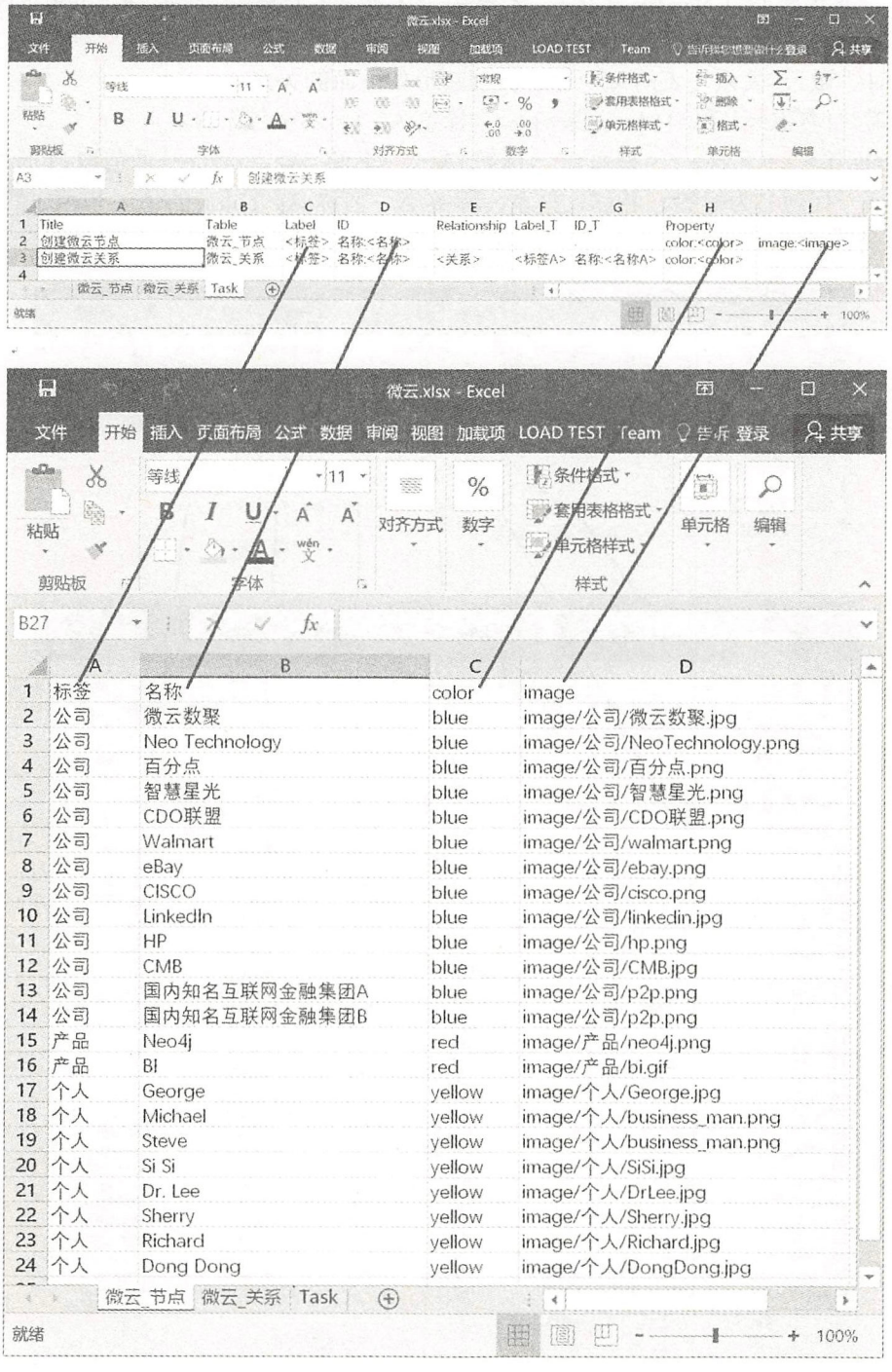


图 9-44　创建节点规则映射图

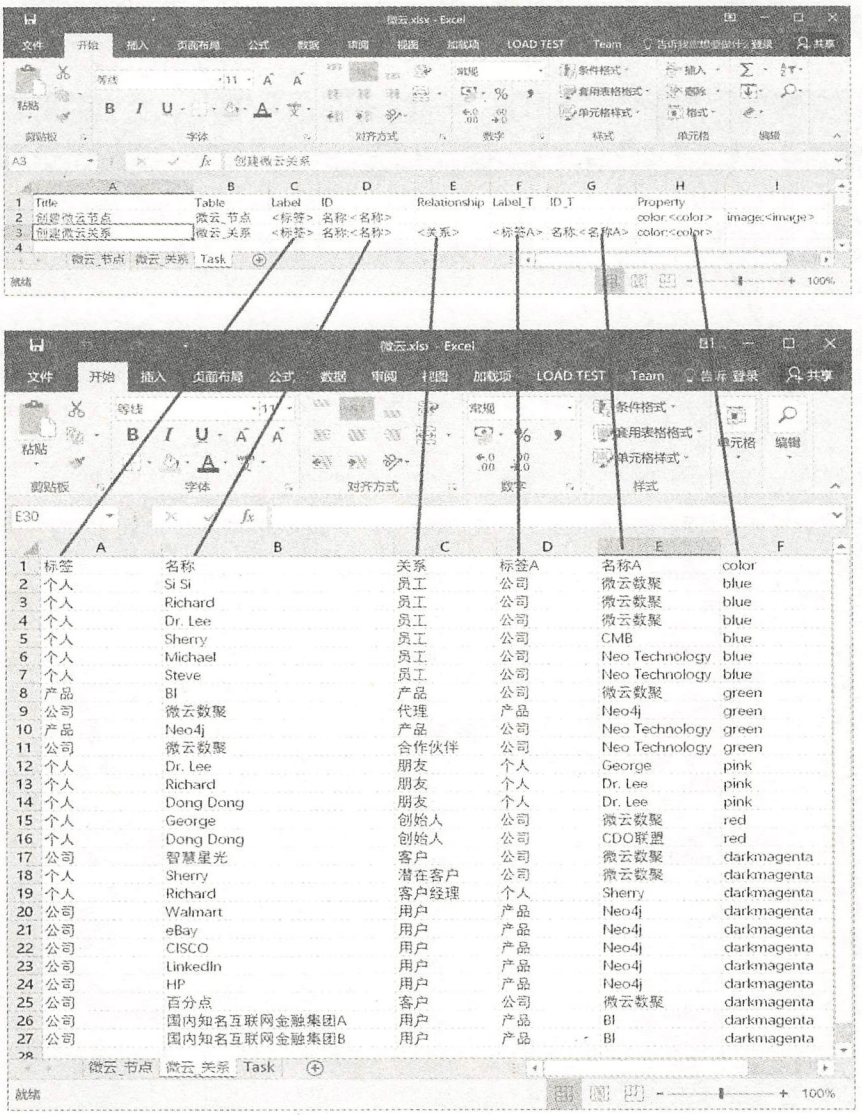
第二行定义“创建关系”规则，数据来源于“微云_关系”表，该规则将遍历“微云_关系”表的每一行，针对每一行创建一个关系。其首节点的标签名称来源于<标签>列，ID 属性

为“名称”，其值来源于<名称>列，其目标节点的标签名称来源于<标签 A>列，ID 属性为“名称”，其值来源于<名称 A>列，关系类型来源于<关系>列，属性 color 的值来源于<color>列。

即针对“微云_关系”表的每一行，将创建形如下面的关系：

(:<标签> {名称:<名称>})-[:<关系>]-> (:<标签 A> {名称:<名称 A>})

其中<标签>、<名称>、<关系>、<标签 A>、<名称 A>、<color>的值取自于“微云_关系”表中每一行列名为标签、名称、关系、标签 A、名称 A、color 的值，其相应的映射关系如图 9-45 所示。



	A	B	C	D	E	F
1	标签	名称	关系	标签A	名称A	color
2	个人	Si Si	员工	公司	微云数聚	blue
3	个人	Richard	员工	公司	微云数聚	blue
4	个人	Dr. Lee	员工	公司	微云数聚	blue
5	个人	Sherry	员工	公司	CMB	blue
6	个人	Michael	员工	公司	Neo Technology	blue
7	个人	Steve	员工	公司	Neo Technology	blue
8	产品	BI	产品	公司	微云数聚	green
9	公司	微云数聚	代理	产品	Neo4j	green
10	产品	Neo4j	产品	公司	Neo Technology	green
11	公司	微云数聚	合作伙伴	公司	Neo Technology	green
12	个人	Dr. Lee	朋友	个人	George	pink
13	个人	Richard	朋友	个人	Dr. Lee	pink
14	个人	Dong Dong	朋友	个人	Dr. Lee	pink
15	个人	George	创始人	公司	微云数聚	red
16	个人	Dong Dong	创始人	公司	CDO联盟	red
17	公司	智慧星光	客户	公司	微云数聚	darkmagenta
18	个人	Sherry	潜在客户	公司	微云数聚	darkmagenta
19	个人	Richard	客户经理	个人	Sherry	darkmagenta
20	公司	Walmart	用户	产品	Neo4j	darkmagenta
21	公司	eBay	用户	产品	Neo4j	darkmagenta
22	公司	CISCO	用户	产品	Neo4j	darkmagenta
23	公司	LinkedIn	用户	产品	Neo4j	darkmagenta
24	公司	HP	用户	产品	Neo4j	darkmagenta
25	公司	百分点	客户	公司	微云数聚	darkmagenta
26	公司	国内知名互联网金融集团A	用户	产品	BI	darkmagenta
27	公司	国内知名互联网金融集团B	用户	产品	BI	darkmagenta
28						

图 9-45　创建节点规则映射图

附录A

Neo4j 配置设置

本附录包含 Neo4j 配置设置的完整参考，适用于设置 neo4j.conf 文件。

1	**browser.allow_outgoing_connections**
说　明	允许 Neo4j 浏览器传出连接
有效值	布尔值
默认值	TRUE

2	**browser.credential_timeout**
说　明	Neo4j 浏览器登录用户的超时时间（即最长闲置时间）。设置为 0 表示没有超时限制
有效值	时间长度（有效单位为 ms（毫秒）、s（秒）、m（分钟）；默认单位是 s）
默认值	0

3	**browser.remote_content_hostname_whitelist**
说　明	允许从 Neo4j 浏览器获取内容的主机白名单
有效值	字符串
默认值	http://guides.neo4j.com,https://guides.neo4j.com,http://localhost,https://localhost

4	**browser.retain_connection_credentials**
说　明	Neo4j 浏览器是否存储用户凭证
有效值	布尔值
默认值	TRUE

5	**causal_clustering.array_block_id_allocation_size**
说　明	当 ARRAY_BLOCK ID 耗尽时，核心服务器创建 ID 分配请求的尺寸。值越大意味着请求频率越低，但在遇到崩溃事件时，也会产生更多的未使用的 ID 和磁盘空间
有效值	整数
默认值	1024

6	**causal_clustering.catchup_batch_size**
说　明	赶上主服务器进度时的最大批量大小（以条目为单位）
有效值	整数
默认值	64

7	**causal_clustering.cluster_allow_reads_on_followers**
说　明	配置 dbms.cluster.routing.getServers() 过程的返回结果是包含读端点还是只返回读副本。当集群中没有读副本时，将忽略本设置，返回读端点
有效值	布尔值
默认值	FALSE

8	**causal_clustering.cluster_routing_ttl**
说　明	驱动程序应该从 dbms.cluster.routing.getServers()过程缓存数据的时间长度
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s），最小值为 1000
默认值	300

9	**causal_clustering.cluster_topology_refresh**
说　明	为刷新当前服务器的拓扑视图而扫描集群的时间间隔
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s），最小值为 1000
默认值	5

10	**causal_clustering.disable_middleware_logging**
说　明	是否开启网络中间件转储日志
有效值	布尔值
默认值	TRUE

11	**causal_clustering.discovery_advertised_address**
说　明	广播集群成员发现管理通信
有效值	广播套接字地址
默认值	localhost:5000

12	**causal_clustering.discovery_listen_address**
说　明	绑定集群成员发现管理通信的主机和端口
有效值	侦听套接字地址
默认值	localhost:5000

13	**causal_clustering.expected_core_cluster_size**
说　明	集群中核心服务器的预期数量
有效值	整数
默认值	3

14	**causal_clustering.global_session_tracker_state_size**
说　明	全局会话追踪器状态文件轮换之前的最大文件尺寸（以条目为单位）
有效值	整数
默认值	1000

15	**causal_clustering.id_alloc_state_size**
说　明	ID 分配文件轮换之前的最大文件尺寸（以条目为单位）
有效值	整数
默认值	1000

16	**causal_clustering.initial_discovery_members**
说　明	要加入集群的其他成员列表（以逗号分隔）
有效值	由逗号分隔的广播套接字地址列表
强制的	强制性的设置项

17	**causal_clustering.join_catch_up_timeout**
说　明	新成员赶上主服务器进度的超时时间
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	600

18	**causal_clustering.label_token_id_allocation_size**
说　明	当 LABEL_TOKEN ID 耗尽时，核心服务器创建 ID 分配请求的尺寸。值越大意味着请求频率越低，但在遇到崩溃事件时，也会产生更多的未使用的 ID 和磁盘空间
有效值	整数
默认值	32

19	**causal_clustering.label_token_name_id_allocation_size**
说　明	当 LABEL_TOKEN_NAME ID 耗尽时，核心服务器创建 ID 分配请求的尺寸。值越大意味着请求频率越低，但在遇到崩溃事件时，也会产生更多的未使用的 ID 和磁盘空间
有效值	整数
默认值	1024

20	**causal_clustering.last_applied_state_size**
说　明	存储文件轮换之前的最大文件尺寸（以条目为单位）
有效值	整数
默认值	1000

21	**causal_clustering.leader_election_timeout**
说　明	如果在该时间限制内没有收到消息，则会发起新的 Leader 选举
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	7

22	**causal_clustering.log_shipping_max_lag**
说　明	日志传输暂停之前允许的最大延迟（以条目为单位）
有效值	整数
默认值	256

23	**causal_clustering.neostore_block_id_allocation_size**
说　明	当 NEOSTORE_BLOCK ID 耗尽时，核心服务器创建 ID 分配请求的尺寸。值越大意味着请求频率越低，但在遇到崩溃事件时，也会产生更多的未使用的 ID 和磁盘空间
有效值	整数
默认值	1024

24	**causal_clustering.node_id_allocation_size**
说　明	当 NODE ID 耗尽时，核心服务器创建 ID 分配请求的尺寸。值越大意味着请求频率越低，但在遇到崩溃事件时，也会产生更多的未使用的 ID 和磁盘空间
有效值	整数
默认值	1024

25	**causal_clustering.node_labels_id_allocation_size**
说　明	当 NODE_LABELS ID 耗尽时，核心服务器创建 ID 分配请求的尺寸。值越大意味着请求频率越低，但在遇到崩溃事件时，也会产生更多的未使用的 ID 和磁盘空间
有效值	整数
默认值	1024

26	**causal_clustering.outgoing_queue_size**
说　明	等待发送到集群中其他服务器的消息数量
有效值	整数
默认值	64

27	**causal_clustering.property_id_allocation_size**
说　明	当 PROPERTY ID 耗尽时，核心服务器创建 ID 分配请求的尺寸。值越大意味着请求频率越低，但在遇到崩溃事件时，也会产生更多的未使用的 ID 和磁盘空间
有效值	整数
默认值	1024

28	**causal_clustering.property_key_token_id_allocation_size**
说　明	当 PROPERTY_KEY_TOKEN ID 耗尽时，核心服务器创建 ID 分配请求的尺寸。值越大意味着请求频率越低，但在遇到崩溃事件时，也会产生更多的未使用的 ID 和磁盘空间
有效值	整数
默认值	32

29	**causal_clustering.property_key_token_name_id_allocation_size**
说　明	当 PROPERTY_KEY_TOKEN_NAME ID 耗尽时，核心服务器创建 ID 分配请求的尺寸。值越大意味着请求频率越低，但在遇到崩溃事件时，也会产生更多的未使用的 ID 和磁盘空间
有效值	整数
默认值	1024

30	**causal_clustering.pull_interval**
说　明	从核心服务器获取更新的时间间隔
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	1

31	**causal_clustering.raft_advertised_address**
说　明	RAFT 集群的广播主机名/IP 地址和端口
有效值	广播套接字地址
默认值	localhost:7000

32	**causal_clustering.raft_listen_address**
说　明	用于 RAFT 集群侦听的网络接口和端口
有效值	侦听套接字地址
默认值	localhost:7000

33	**causal_clustering.raft_log_implementation**
说　明	RAFT 集群的日志实现
有效值	字符串
默认值	SEGMENTED

34	**causal_clustering.raft_log_prune_strategy**
说　明	RAFT 集群的日志修剪策略
有效值	字符串
默认值	1g size

35	**causal_clustering.raft_log_pruning_frequency**
说　明	RAFT 集群的日志修剪频率
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	600

36	**causal_clustering.raft_log_reader_pool_size**
说　明	RAFT 集群的日志读取器池的尺寸
有效值	整数
默认值	8

37	**causal_clustering.raft_log_rotation_size**
说　明	RAFT 集群的日志轮换尺寸
有效值	字节尺寸（有效值为 k、m、g、K、M、G 的倍数），最小值为 1024
默认值	262144000

38	**causal_clustering.raft_membership_state_size**
说　明	会员状态文件轮换之前的最大文件尺寸（以条目为单位）
有效值	整数
默认值	1000

39	**causal_clustering.raft_messages_log_enable**
说　明	启用或禁用与 RAFT 协议有关的所有网络消息的转储
有效值	布尔值
默认值	FALSE

40	**causal_clustering.raft_term_state_size**
说　明	时间周期状态文件轮换之前的最大文件尺寸（以条目为单位）
有效值	整数
默认值	1000

41	**causal_clustering.raft_vote_state_size**
说　明	投票状态文件轮换之前的最大文件尺寸（以条目为单位）
有效值	整数
默认值	1000

42	**causal_clustering.read_replica_time_to_live**
说　明	读副本被判断为不可用之前的生存时间
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s），最小值为 60000
默认值	60

43	**causal_clustering.refuse_to_be_leader**
说　明	防止当前实例主动成为 RAFT 集群的 leader 角色。默认值是 FALSE，只能在特殊情况下由专家用户使用，使用它可能会导致集群的可用性降低
有效值	布尔值
默认值	FALSE

44	**causal_clustering.relationship_group_id_allocation_size**
说　明	当 RELATIONSHIP_GROUP ID 耗尽时，核心服务器创建 ID 分配请求的尺寸。值越大意味着请求频率越低，但在遇到崩溃事件时，也会产生更多的未使用的 ID 和磁盘空间
有效值	整数
默认值	1024

45	**causal_clustering.relationship_id_allocation_size**
说　明	当 RELATIONSHIP ID 耗尽时，核心服务器创建 ID 分配请求的尺寸。值越大意味着请求频率越低，但在遇到崩溃事件时，也会产生更多的未使用的 ID 和磁盘空间
有效值	整数
默认值	1024

46	**causal_clustering.relationship_type_token_id_allocation_size**
说　明	当 RELATIONSHIP_TYPE_TOKEN ID 耗尽时，核心服务器创建 ID 分配请求的尺寸。值越大意味着请求频率越低，但在遇到崩溃事件时，也会产生更多的未使用的 ID 和磁盘空间
有效值	整数
默认值	32

47	**causal_clustering.relationship_type_token_name_id_allocation_size**
说　明	当 RELATIONSHIP_TYPE_TOKEN_NAME ID 耗尽时，核心服务器创建 ID 分配请求的尺寸。值越大意味着请求频率越低，但在遇到崩溃事件时，也会产生更多的未使用的 ID 和磁盘空间
有效值	整数
默认值	1024

48	**causal_clustering.replicated_lock_token_state_size**
说　明	复制锁令牌状态文件轮换之前的最大文件尺寸（以条目为单位）
有效值	整数
默认值	1000

49	**causal_clustering.schema_id_allocation_size**
说　明	当 SCHEMA ID 耗尽时，核心服务器创建 ID 分配请求的尺寸。值越大意味着请求频率越低，但在遇到崩溃事件时，也会产生更多的未使用的 ID 和磁盘空间
有效值	整数
默认值	1024

50	**causal_clustering.state_machine_apply_max_batch_size**
说　明	在状态机中操作应用期间要批量处理的最大操作数量
有效值	整数
默认值	16

51	**causal_clustering.state_machine_flush_window_size**
说　明	在状态机将数据刷新到磁盘之前要处理的操作数量
有效值	整数
默认值	4096

52	**causal_clustering.string_block_id_allocation_size**
说　明	当 STRING_BLOCK ID 耗尽时，核心服务器创建 ID 分配请求的尺寸。值越大意味着请求频率越低，但在遇到崩溃事件时，也会产生更多的未使用的 ID 和磁盘空间
有效值	整数
默认值	1024

53	**causal_clustering.transaction_advertised_address**
说　明	事务传送服务器的广播主机名/IP 地址和端口
有效值	广播套接字地址
默认值	localhost:6000

54	**causal_clustering.transaction_listen_address**
说　明	事务传送服务器侦听用的网络接口和端口
有效值	侦听套接字地址
默认值	localhost:6000

55	**causal_clustering.unknown_address_logging_throttle**
说　明	记录未知集群成员地址的节流限制
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	10

56	**cypher.default_language_version**
说　明	指定默认解析器（语言版本）
有效值	下列之一：2.3，3.0，3.1，default
默认值	Default

57	**cypher.forbid_exhaustive_shortestpath**
说　明	此设置与性能优化相关。将其设置为 true，则在使用“shortestPath”函数查询时，若无答案就会尽快终止，不去试图为找到答案而花费更多时间（即使没找到路径）。对于大多数查询而言，“shortestPath”算法将很快返回正确的答案。然而，在某些情况下，即使存在快速双向宽度优先搜索算法也不会发现结果。当 WHERE 子句中应用于“shortestPath”的谓词不能应用于遍历的每个步骤时，都可能发生这种情况，因此只能应用于整个路径。当查询计划器检测到这些特殊情况时，如果快速算法找不到路径，它将计划执行详尽的深度优先搜索。然而，穷举搜索可能比快速算法慢几个数量级。如果查询尽快终止是至关重要的，则建议将此选项设置为 true，这意味着 Neo4j 将永远不会考虑使用穷举搜索的最短路径查询。但请注意，如果找不到路径，运行时则会抛出错误，交给应用程序来处理
有效值	布尔值
默认值	FALSE

58	**cypher.forbid_shortestpath_common_nodes**
说　明	此设置与性能优化相关。当起始和终止节点相同时，“shortestPath”算法是不起作用的。当设置为 false 时，不会返回任何路径；为 true 的默认值则会抛出异常。如果在笛卡尔乘积之后执行“shortestPath”搜索，传递给“shortestPath”的某些行可能具有相同的起始和终止节点，则可能会发生这种情况。如果自信不会遇到这种异常，并且可以接受这种结果，则可以将其设置为 false。如果不能接受可能的结果丢失，并且真正希望得到两个公共节点之间的最短路径，则使用标准的 Cypher 可变长度模式表达式重写查询，再按路径长度排序并限制为一个结果
有效值	布尔值
默认值	TRUE

59	**cypher.hints_error**
说　明	设置此值可指定当 Cypher 计划器或运行时提示无法实现时的行为。如果为 true，则“不一致”将导致错误，否则只会产生警告
有效值	布尔值
默认值	FALSE

60	**cypher.min_replan_interval**
说　明	查询计划的最短生命周期，超过这个时间后，查询将会被重新计划
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	10

61	**cypher.planner**
说　明	为默认语言版本指定默认的计划器
有效值	下列之一：COST、RULE、default
默认值	Default

62	**cypher.statistics_divergence_threshold**
说　明	判断计划是否过时的阈值。如果用于创建计划的任何基本统计信息的变化超过该阈值，则被判断为过时并将重新计划。设置为 0 表示总是重新计划，1 表示永不重新计划
有效值	浮点值，最小值为 0.0，最大值为 1.0
默认值	0.75

63	**dbms.active_database**
说　明	要加载的数据库的名称
有效值	字符串
默认值	graph.db

64	**dbms.allow_format_migration**
说　明	是否允许库升级，以防数据库的当前版本针对较旧的库版本启动。将此设置为 true 并不能保证升级成功，只允许进行升级
有效值	布尔值
默认值	FALSE

65	**dbms.backup.address**
说　明	侦听联机备份服务器
有效值	主机名和端口
默认值	127.0.0.1:6362-6372

66	**dbms.backup.enabled**
说　明	启用对运行联机备份的支持
有效值	布尔值
默认值	TRUE

67	**dbms.checkpoint.interval.time**
说　明	配置检查点之间的时间间隔。数据库检查点的频率不会高于此间隔（除非其他事件触发）。但如果检查点的执行时间长于此间隔，则实际的检查点可能会比此间隔更少。检查点是事务日志中的一个可恢复点。检查点间隔越长，通常也意味着遇到崩溃时完成恢复所需的时间也越长。而另一方面，较长的检查点间隔可以减少数据库所在系统的 I/O 负载，因为每个检查点必然会有一次对所有库文件的强制刷新
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	300

68	**dbms.checkpoint.interval.tx**
说　明	配置检查点之间的事务间隔。数据库检查点的频率不会高于此设置（除非其他事件触发）。但如果检查点的执行时间长于此间隔，则实际的检查点可能会比此间隔更少。检查点是事务日志中的一个可恢复点。检查点间隔越长，通常也意味着遇到崩溃时完成恢复所需的时间越长。而另一方面，较长的检查点间隔也可以减少数据库所在系统的 I/O 负载，因为每个检查点必然会有一次对所有库文件的强制刷新。默认值为“100000”，表示每 10 万次有一个检查点事务
有效值	整数，最小值为 1
默认值	100000

69	**dbms.checkpoint.iops.limit**
说　明	限制后台检查点进程每秒将消耗的 IO 次数。此设置是建议性的，Neo4j 社区版将忽略这个设置。Neo4j 企业版则尽最大努力来支持它。IO 每次写入 8 KiB（大部分是顺序的）。以这种方式限制写 IO 将为 IO 子系统留下更多带宽来服务于随机读取 IO，这对于不能将数据库完全填入内存的查询响应时间很重要。此设置的唯一缺点是，一旦遇到数据库或系统崩溃，检查点的时间越长，恢复时间会越长。这个数字越小，IO 压力就越小，而检查点次数就越多。也可以注释掉此配置，以完全消除限制，且让检查点刷新数据与硬件一样快。将其设置为 -1 则禁用 IOPS 限制
有效值	整数
默认值	1000

70	**dbms.connectors.default_advertised_address**
说　明	服务器用于向其连接器发布自身的默认主机名或 IP 地址。要为特定连接器发布特定主机名或 IP 地址，需要指定特定连接器的 advertised_address 属性
有效值	字符串
默认值	Localhost

71	**dbms.connectors.default_listen_address**
说　明	监听传入连接的默认网络接口。若要监听所有接口上的连接，请使用“0.0.0.0”。若要将特定连接器绑定到特定的网络接口，请指定特定连接器的 listen_address 属性
有效值	字符串
默认值	Localhost

72	**dbms.directories.certificates**
说　明	用于存储 Neo4j 的 TLS 连接证书的目录。证书文件必须命名为 neo4j.cert 和 neo4j.key
有效值	文件系统路径；相对路径根据安装根 <neo4j-home> 进行解析
默认值	certificates

73	**dbms.directories.data**
说　明	data 目录的路径。多个 Neo4j 安装的配置不能指向同一个 data 目录
有效值	文件系统路径；相对路径根据安装根 <neo4j-home> 进行解析
默认值	Data

74	**dbms.directories.import**
说　明	设置与 Cypher LOAD CSV 子句一起使用的文件 URL 的根目录。必须设置为单个目录，限制成只能访问该目录及其子目录中的文件
有效值	文件系统路径；相对路径根据安装根 <neo4j-home> 进行解析

75	**dbms.directories.lib**
说　明	lib 目录的路径
有效值	文件系统路径；相对路径根据安装根 <neo4j-home> 进行解析
默认值	Lib

76	**dbms.directories.logs**
说　明	logs 目录的路径
有效值	文件系统路径；相对路径根据安装根 <neo4j-home> 进行解析
默认值	Logs

77	**dbms.directories.metrics**
说　明	CSV 文件的目标位置，即存放所有 CSV 文件的目录路径
有效值	文件系统路径；相对路径根据安装根 <neo4j-home> 进行解析
默认值	Metrics

78	**dbms.directories.plugins**
说　明	数据库插件目录的位置，即存放包含数据库过程的已编译的 Java JAR 文件的目录路径
有效值	文件系统路径；相对路径根据安装根 <neo4j-home> 进行解析
默认值	plugins

79	**dbms.directories.run**
说　明	运行目录的路径。此目录包含 Neo4j 的运行时状态，比如在后台运行时的 pidfile，该文件在 Neo4j 启动时被创建，在停止时被删除。它可以被存放在诸如 tmpfs 的内存文件系统中
有效值	文件系统路径；相对路径根据安装根 <neo4j-home> 进行解析
默认值	Run

80	**dbms.ids.reuse.types.override**
说　明	要重复使用的 id 类型（逗号分隔）的指定名称。目前只支持“节点”和“关系”类型
有效值	由逗号分隔的 NODE 和 RELATIONSHIP 列表
默认值	[RELATIONSHIP, NODE]

81	**dbms.import.csv.legacy_quote_escaping**
说　明	选择是否按照标准 https://tools.ietf.org/html/rfc4180 解释使用 LOAD CSV 加载的 CSV 文件中的转义引号。将此设置为 false 将使用标准，即将重复的引号“""”解释为一个内置的引号；而 true 将使用 Neo4j 3.0 和 3.1 中原来的约定，允许在字符串中用反斜杠包含引号
有效值	布尔值
默认值	TRUE

82	**dbms.index_sampling.background_enabled**
说　明	启用或禁用后台索引抽样
有效值	布尔值
默认值	TRUE

83	**dbms.index_sampling.buffer_size**
说　明	索引抽样所用的缓存区尺寸。从 Neo4j 3.0.3 起，此设置不再适用。请使用 bms.index_sampling.sample_size_limit
有效值	字节尺寸（有效值为 k、m、g、K、M、G 的倍数），最小值为 1048576，最大值为 2147483647
默认值	67108864
过时的	该配置设置项已过时

84	**dbms.index_sampling.sample_size_limit**
说　明	索引抽样块的尺寸限制
有效值	整数，最小值为 1048576，最大值为 2147483647
默认值	8388608

85	**dbms.index_sampling.update_percentage**
说　明	触发给定索引抽样之前，索引更新占总索引尺寸的百分比
有效值	整数，最小值为 0
默认值	5

86	**dbms.index_searcher_cache_size**
说　明	开放 Lucene 指数搜索的最大数量
有效值	整数，最小值为 1
默认值	2147483647

87	**dbms.logs.debug.level**
说　明	调试日志级别的阈值
有效值	下列之一：DEBUG、INFO、WARN、ERROR、NONE
默认值	INFO

88	**dbms.logs.debug.rotation.delay**
说　明	调试日志最后一次轮换之后可能会再次轮换的最短时间间隔
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	300

89	**dbms.logs.debug.rotation.keep_number**
说　明	调试日志的最大历史文件数
有效值	整数，最小值为 1
默认值	7

90	**dbms.logs.debug.rotation.size**
说　明	调试日志的轮换阈值
有效值	字节尺寸（有效值为 k、m、g、K、M、G 的倍数），最小值为 0，最大值为 9223372036854775807
默认值	20971520

91	**dbms.logs.gc.enabled**
说　明	开启 GC（Garbage Collection）日志记录
有效值	布尔值
默认值	FALSE

92	**dbms.logs.gc.options**
说　明	GC 日志选项
有效值	字符串
默认值	-XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCApplicationStoppedTime -XX:+PrintPromotionFailure -XX:+PrintTenuringDistribution

93	**dbms.logs.gc.rotation.keep_number**
说　明	要保留的 GC 日志数
有效值	整数
默认值	5

94	**dbms.logs.gc.rotation.size**
说　明	要保留的 GC 日志数
有效值	字节尺寸（有效值为 k、m、g、K、M、G 的倍数），最小值为 0，最大值为 9223372036854775807
默认值	20971520

95	**dbms.logs.http.enabled**
说　明	启用 HTTP 请求日志
有效值	布尔值
默认值	FALSE

96	**dbms.logs.http.rotation.keep_number**
说　明	要保留的 HTTP 日志数
有效值	整数
默认值	5

97	**dbms.logs.http.rotation.size**
说　明	要保留的每个 HTTP 日志的尺寸
有效值	字节尺寸（有效值为 k、m、g、K、M、G 的倍数），最小值为 0，最大值为 9223372036854775807
默认值	20971520

98	**dbms.logs.query.enabled**
说　明	记录执行时间超过 dbms.logs.query.threshold 所配置的阈值的查询。日志条目将写入 Logs 目录中的 filequery.log。此功能在 Neo4j 企业版中可用
有效值	布尔值
默认值	FALSE

99	**dbms.logs.query.parameter_logging_enabled**
说　明	执行查询的日志参数需要比配置的阈值更长的时间
有效值	布尔值
默认值	TRUE

100	**dbms.logs.query.rotation.keep_number**
说　明	查询日志的最大历史文件数
有效值	整数，最小值为 1
默认值	7

101	**dbms.logs.query.rotation.size**
说　明	查询日志自动轮换的文件尺寸（以字节为单位）。如果设置为零，则不会发生轮换。接受二进制后缀 k、m 或者 g
有效值	字节尺寸（有效值为 k、m、g、K、M、G 的倍数），最小值为 0，最大值为 9223372036854775807
默认值	20971520

102	**dbms.logs.query.threshold**
说　明	如果启用了查询日志，当执行查询所花费的时间超过此阈值时，则会记录该查询。默认为 0 秒，即所有查询都被记录
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	0

103	**dbms.logs.security.level**
说　明	安全日志级别阈值
有效值	下列之一：DEBUG、INFO、WARN、ERROR、NONE
默认值	INFO

104	**dbms.logs.security.rotation.delay**
说　明	安全日志最后轮换之后到可能会再次轮换前的最短时间间隔
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	300

105	**dbms.logs.security.rotation.keep_number**
说　明	安全日志的最大历史文件数
有效值	整数，最小值为 1
默认值	7

106	**dbms.logs.security.rotation.size**
说　明	安全日志轮换的阈值
有效值	字节尺寸（有效值为 k、m、g、K、M、G 的倍数），最小值为 0，最大值为 9223372036854775807
默认值	20971520

107	**dbms.memory.pagecache.size**
说　明	用于映射库文件的内存量（后缀表示单位，“k”为千字节，“m”为兆字节，“g”为千兆字节）。如果 Neo4j 在专用服务器上运行，那么通常建议为操作系统留下大约 2～4 GB 的资源，给予 JVM 足够的堆来保存所有的事务状态、查询上下文及页面缓存。如果没有配置页面缓存内存，则基于可用的系统资源计算启发式设置
有效值	字节尺寸（有效值为 k、m、g、K、M、G 的倍数），最小值为 245760

108	**dbms.memory.pagecache.swapper**
说　明	指定要用于分页 IO 的页面交换器。本设置仅适用于与专有存储技术的集成
有效值	字符串

109	**dbms.mode**
说　明	配置数据库的操作模式：“SINGLE”用于独立运行，“HA”作为 HA 集群中的成员运行，“ARBITER”用于 HA 集群中没有数据库的集群成员运行，“CORE”用于作为因果集群的核心成员运行，“READ_REPLICA”作为因果集群的读取副本成员运行
有效值	字符串
默认值	SINGLE

110	**dbms.query_cache_size**
说　明	被缓存的 Cypher 查询执行计划的数量
有效值	整数，最小值为 0
默认值	1000

111	**dbms.read_only**
说　明	只允许从该 Neo4j 实例进行读取操作。此模式仍然需要对目录具有写访问权限以进行锁定
有效值	布尔值
默认值	FALSE

112	**dbms.record_format**
说　明	数据库记录格式。企业版独有。有效值为：standard，high_limit。默认值为 standard
有效值	字符串
默认值	Standard

113	**dbms.relationship_grouping_threshold**
说　明	判断节点是密集的关系计数阈值
有效值	整数，最小值为 1
默认值	50

114	**dbms.rest.transaction.idle_timeout**
说　明	REST 终端中的空闲事务超时设置
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	60

115	**dbms.security.allow_csv_import_from_file_urls**
说　明	确定在使用 LOAD CSV 加载数据时，Cypher 是否使用文件 URL。将此值设置为 false，则从文件系统加载数据的 CSV 子句将导致 Neo4j 失败
有效值	布尔值
默认值	TRUE

116	**dbms.security.auth_cache_max_capacity**
说　明	认证和授权（分别的）缓存的最大容量
有效值	整数
默认值	10000

117	**dbms.security.auth_cache_ttl**
说　明	使用外部验证提供程序（LDAP 或插件）时，用于缓存身份验证和授权信息的 TTL（Time To Live）时间。将 TTL 设置为 0 将禁用验证缓存。在使用 LDAP 认证提供程序时禁用缓存，需要使用 LDAP 系统账户来解决授权信息
有效值	时间长度（有效单位为 ms，s，m；默认单位是 s）
默认值	600

118	**dbms.security.auth_enabled**
说　明	启用密码验证以访问 Neo4j
有效值	布尔值
默认值	FALSE

119	**dbms.security.auth_provider**
说　明	包含用户和角色的身份验证和授权提供程序。可以是内置的 native 或 ldap 提供程序，也可以是外部提供的插件，其自定义名称前面冠以前缀 plugin-，即 plugin-<AUTH_PROVIDER_NAME>
有效值	字符串
默认值	native

120	**dbms.security.causal_clustering_status_auth_enabled**
说　明	需要授权以访问因果集群状态端点
有效值	布尔值
默认值	TRUE

121	**dbms.security.ha_status_auth_enabled**
说　明	需要授权以访问 HA 状态端点
有效值	布尔值
默认值	TRUE

122	**dbms.security.http_authorization_classes**
说　明	Neo4j 使用的自定义安全规则列表（以逗号分隔）
有效值	由逗号分隔的字符串列表
默认值	[]

123	**dbms.security.ldap.authentication.cache_enabled**
说　明	确定是否应缓存通过 LDAP 服务器进行身份验证的结果。这种缓存可用来减少已成功验证的用户再次通过网络进行 LDAP 请求的数量。只要用户还处于存活状态（seedbms.security.auth_cache_ttl），就可利用已有的缓存条目（而不是通过 LDAP 服务器）来进行身份验证。将此设置为 true，意味着 Neo4j 需要缓存凭证的散列版本才能执行凭证匹配
有效值	布尔值
默认值	TRUE

124	**dbms.security.ldap.authentication.mechanism**
说　明	LDAP 认证机制，可以是 simple 或由 JNDI 支持的 SASL 机制之一，例如 DIGEST-MD5。Simple 是基本用户名和密码的认证，SASL 则是更先进的认证机制。有关更多详细信息，请参阅 RFC 2251 LDAPv3 文档
有效值	字符串
默认值	Simple

125	**dbms.security.ldap.authentication.user_dn_template**
说　明	LDAP 用户 DN 模板。LDAP 对象由其可区别的名称（DN）引用，用户 DN 是 LDAP 完全限定的唯一用户标识符。此设置用于生成与登录时使用身份验证令牌提交的用户主体中的 LDAP 目录的模式一致的 LDAP DN。特殊令牌 {0} 是用户主体将被替换为 DN 的占位符
有效值	字符串
默认值	uid={0},ou=users,dc=example,dc=com

126	**dbms.security.ldap.authorization.group_membership_attributes**
说　明	在启用 LDAP 授权时，映射到角色组的用户对象上的属性名称列表
有效值	由逗号分隔的字符串列表
默认值	[memberOf]

127	**dbms.security.ldap.authorization.group_to_role_mapping**
说　明	从 LDAP 组名称到 Neo4j 角色名称的授权映射。映射的格式是键值对列表（以分号分隔），其中，键是 LDAP 组名称，值是角色名称列表（以逗号分隔）。例如：group1 = role1; group2 = role2; group3 = role3，role4，role5。 为映射更易读懂，也可以在组名称中使用空格和引号，例如： dbms.security.ldap.authorization.group_to_role_mapping=\ "cn=Neo4j Read Only,cn=users,dc=example,dc=com" = reader;\ "cn=Neo4j Read-Write,cn=users,dc=example,dc=com" = publisher;\ "cn=Neo4j Schema Manager,cn=users,dc=example,dc=com" = architect;\ "cn=Neo4j Administrator,cn=users,dc=example,dc=com" = admin.
有效值	字符串

128	**dbms.security.ldap.authorization.system_password**
说　明	用于授权搜索的 LDAP 系统账户密码（当 dbms.security.ldap.authorization.use_system_accountis 为 true 时）。
有效值	字符串

129	**dbms.security.ldap.authorization.system_username**
说　明	用于授权搜索的 LDAP 系统账户用户名（当 dbms.security.ldap.authorization.use_system_account 为 true 时）。请注意，dbms.security.ldap.authentication.user_dn_template 不会使用此用户名，必须指定一个完整的 DN
有效值	字符串

130	**dbms.security.ldap.authorization.use_system_account**
说　明	使用系统账户而不是用户自己的账户来执行授权信息的 LDAP 搜索。 如果设置为 false（默认），则使用与用户自己的账户绑定的 LDAP 上下文后，直接执行对组成员资格的搜索。映射的角色将在 dbms.security.auth_cache_ttl 期间缓存，过期后需要重新认证。为了避免频繁地重新验证会话，可将这个选项设置为相对较长的身份验证缓存到期时间。 注意：仅当用户被允许在目录中搜索自己的组成员身份属性时，此选项才会起作用。 如果设置为 true，则将使用具有对该目录中所有用户的读访问权限的特殊系统账户执行搜索。同时需要使用 dbms.security.ldap.authorization.system_username 和 dbms.security.ldap.authorization.system_password 设置来指定用户名和密码。请注意，此账户只需要对 LDAP 目录的相关部分进行读取访问，不需要具有对 Neo4j 或任何其他系统的访问权限
有效值	布尔值
默认值	FALSE

131	**dbms.security.ldap.authorization.user_search_base**
说　明	启用 LDAP 授权时，用于搜索用户对象的基础对象或命名上下文的名称。常见的情况是与 dbms.security.ldap.authentication.user_dn_template 的最后一部分相匹配
有效值	字符串
默认值	ou=users,dc=example,dc=com

132	**dbms.security.ldap.authorization.user_search_filter**
说　明	LDAP 搜索过滤器，用于在启用 LDAP 授权时搜索用户主体。过滤器应包含用于替换用户主体的占位符令牌 {0}
有效值	字符串
默认值	(&(objectClass=*)(uid={0}))

133	**dbms.security.ldap.connection_timeout**
说　明	建立 LDAP 连接的超时时间。如果在给定时间内无法建立与 LDAP 服务器的连接，则尝试终止。值为 0 表示使用网络协议（即 TCP）的超时值
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	30

134	**dbms.security.ldap.host**
说　明	用于认证和授权的 LDAP 服务器的 URL。设置的格式是 <protocol>://<hostname>:<port>，其中 hostname 是唯一必需的字段。协议支持的值有 ldap（默认值）和 ldaps。ldap 的默认端口是 389，ldaps 的默认端口是 636。例如：ldaps://ldap.example.com:10389
有效值	字符串
默认值	Localhost

135	**dbms.security.ldap.read_timeout**
说　明	LDAP 读取请求（如搜索）的超时。如果 LDAP 服务器在给定的时间内没有响应，请求将被终止。值为 0 表示无限期等待响应
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	30

136	**dbms.security.ldap.referral**
说　明	创建连接时的 LDAP 的 referral 行为。可以是 follow、ignore 或 throw 之一。*follow 则自动遵循任何 referral，* ignore 则忽略任何 referral，* throw 则抛出异常并导致身份验证失败
有效值	字符串
默认值	Follow

137	**dbms.security.ldap.use_starttls**
说　明	使用机会性 TLS 与 LDAP 服务器进行安全通信。首先用 LDAP 服务器发起初始的不安全连接，并在发起身份验证之前发出 STARTTLS 命令来协商到 TLS 的升级连接
有效值	布尔值
默认值	FALSE

138	**dbms.security.log_successful_authentication**
说　明	如果设置为 true，则会将成功的认证事件记录到安全日志中。如果设置为 false，则只记录失败的身份验证事件。如果发现成功事件引起的日志过多且多余，则可设置为 false .
有效值	布尔值
默认值	TRUE

139	**dbms.security.procedures.default_allowed**
说　明	设置可以执行 dbms.security.procedures.roles 未涵盖的所有过程和用户定义函数的默认角色。如果 dbms.security.procedures.default_allowed 设置为空字符串（默认），则将按照与常规 Cypher 语句相同的安全规则执行过程
有效值	字符串
默认值	

140	**dbms.security.procedures.roles**
说　明	本设置提供了比 dbms.security.procedures.default_allowed 设置级别更细的控制，可指定哪些角色可以执行过程。例如：“dbms.security.procedures.roles=apoc.convert.:reader;apoc.load.json:writer;apoc.trigger.add:TriggerHappy”将允许角色 reader 可执行 apoc.convert 命名空间中的所有过程，角色 writer 可执行以 json 开头的 apoc.load 命名空间中的所有过程，角色 TriggerHappy 可执行特定的过程 apoc.trigger.add 。与这些模式不匹配的过程则受 dbms.security.procedures.default_allowed 设置的约束
有效值	字符串
默认值	

141	**dbms.shell.enabled**
说　明	启用 Neo4j Shell 客户端可以登录的远程 shell 服务器
有效值	布尔值
默认值	FALSE

142	**dbms.shell.host**
说　明	shell 远程主机。默认情况下，shell 服务器仅在 loopback 接口上侦听。可以指定任何网络接口的 IP 地址，或使用 0.0.0.0 用于所有接口
有效值	字符串，必须为有效的名称
默认值	127.0.0.1

143	**dbms.shell.port**
说　明	shell 将侦听的端口
有效值	整数，必须为有效的端口号（范围 0～65535）
默认值	1337

144	**dbms.shell.read_only**
说　明	只读模式。只允许读操作
有效值	布尔值
默认值	FALSE

145	**dbms.shell.rmi_name**
说　明	shell 的名称
有效值	字符串，必须为有效的名称
默认值	Shell

146	**dbms.threads.worker_count**
说　明	Neo4j 工作线程的数量。操作系统可能会强制一个比这个值更低的下限
有效值	整数，范围 1～44738
默认值	“处理器数”与 500 之间的最小值

147	**dbms.transaction.timeout**
说　明	完成事务的最长时间间隔
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	0

148	**dbms.tx_log.rotation.retention_policy**
说　明	使 Neo4j 保留逻辑事务日志以便能够备份数据库。可以用于指定修剪逻辑日志的阈值。例如，“10 days”将修剪成仅包含当前时间以前 10 天内的逻辑事务日志，又如“100k txs”将保留 100k 的最新事务并修剪任何较旧的事务
有效值	字符串，必须为 true/false 或为“<数字><可选单位> <类型>”格式。例如 100M size 表示限制逻辑日志空间的磁盘空间为 100MB，又如 200k txs 表示限制事务数量为 200000
默认值	7 days

149	**dbms.tx_log.rotation.size**
说　明	指定逻辑日志自动轮换的文件尺寸。0 表示不会自动根据文件尺寸发生轮换
有效值	字节尺寸（有效值为 k、m、g、K、M、G 的倍数），最小值为 1048576
默认值	262144000

150	**dbms.udc.enabled**
说　明	启用 UDC 扩展
有效值	布尔值
默认值	TRUE

151	**dbms.unmanaged_extension_classes**
说　明	非托管扩展的 <classname>=<mount point> 列表（以逗号分隔）
有效值	<classname>=<mount point>字符串列表（以逗号分隔）
默认值	[]

152	**ha.allow_init_cluster**
说　明	在无法加入集群时，是否允许本实例创建一个集群
有效值	布尔值
默认值	TRUE

153	**ha.branched_data_copying_strategy**
说　明	指定如何订购从服务器上分支数据的处理以及从主服务器复制库的策略。默认值为 copy_then_branch，当与 keep_last 或 keep_none 分支处理策略结合使用时，会产生更安全的分支策略，因为始终都会有一个库存在，即使复制库失败（例如，由于网络故障），该实例也不会没有一个库
有效值	下列之一：branch_then_copy、copy_then_branch
默认值	branch_then_copy

154	**ha.branched_data_policy**
说　明	如何处理分支数据的策略
有效值	下列之一：keep_all、keep_last、keep_none
默认值	keep_all

155	**ha.broadcast_timeout**
说　明	集群中的广播值超时时间。必须考虑 Paxos 算法的端到端持续时间。该值是 ha.join_timeout 和 ha.leave_timeout 设置的默认值
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	30

156	**ha.configuration_timeout**
说　明	在集群连接期间等待从现有集群成员进行配置的超时时间
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	1

157	**ha.data_chunk_size**
说　明	HA 中主服务器和从服务器之间流动的数据块的最大尺寸。尺寸越大，可能的吞吐量会越大，但对带宽的变化也会更敏感，而较小的尺寸将更能接受带宽的变化
有效值	字节尺寸（有效值为 k、m、g、K、M、G 的倍数），最小值为 1024
默认值	2097152

158	**ha.default_timeout**
说　明	集群超时的默认超时。在必要时可用适当的值覆盖这个特定的超时设置。该值是 ha.heartbeat_interval，ha.paxos_timeout 和 ha.learn_timeout 设置的默认值
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	5

159	**ha.election_timeout**
说　明	等待其他成员完成角色选举的超时时间。默认为 ha.paxos_timeout
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	5

160	**ha.heartbeat_interval**
说　明	心跳信息发送的间隔时间。默认为 ha.default_timeout
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	5

161	**ha.heartbeat_timeout**
说　明	等待来自其他实例的心跳时间长度，超过这个时间就怀疑其通信失败。此值反映了网络延迟、预期的垃圾收集暂停，以及可能影响消息发送和处理的其他因素。值越大，主服务器越稳定，但在主服务器发生故障时，也会导致故障转移之前的等待时间更长。该值不能小于 ha.heartbeat_interval 值的两倍，否则会存在频繁的主服务器交换及出现可能的分支数据等高风险
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	40

162	**ha.host.coordination**
说　明	绑定集群管理通信的主机和端口
有效值	主机和端口
默认值	0.0.0.0:5001-5099

163	**ha.host.data**
说　明	绑定 HA 服务器的主机名和端口
有效值	主机名和端口
默认值	0.0.0.0:6001-6011

164	**ha.initial_hosts**
说　明	要加入的集群的其他成员列表（以逗号分隔）
有效值	由逗号分隔的主机名和端口列表
强制的	强制性的设置项

165	**ha.internal_role_switch_timeout**
说　明	在切换到主服务器或从服务器之前，等待状态切换（如事务完成）内部条件的超时时间
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	10

166	**ha.join_timeout**
说　明	加入集群的超时时间。默认为 ha.broadcast_timeout。请注意，如果在集群形成期间超时，则操作员可能有必要重新启动实例
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	30

167	**ha.learn_timeout**
说　明	学习值的超时时间。默认为 ha.default_timeout
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	5

168	**ha.leave_timeout**
说　明	等待集群离开完成的超时时间。默认为 ha.broadcast_timeout
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	30

169	**ha.max_acceptors**
说　明	在同意更改成员资格时涉及的最大服务器数量。在非常大的集群中，集群出现一半故障的概率很低，但要防止一半的故障却非常昂贵。因此，一般都会将此参数设置为小于集群的大小
有效值	整数，最小值为 1
默认值	21

170	**ha.max_channels_per_slave**
说　明	从服务器与主服务器的最大连接数
有效值	整数，最小值为 1
默认值	20

171	**ha.paxos_timeout**
说　明	所有 Paxos 超时的默认值。此设置控制 ha.phase1_timeout、ha.phase2_timeout 和 ha.election_timeout 设置的默认值。如果没有赋值，它将默认为 ha.default_timeout，如果 ha.default_timeout 更改则将隐式更改。这是一个高级参数，只有在 Neo4j 专业服务专门提供建议的情况下才能更改
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	5

172	**ha.phase1_timeout**
说　明	Paxos phase 1 的超时时间。如果没有设置，它将默认为 ha.paxos_timeout，如果 ha.paxos_timeout 更改则将隐式更改。这是一个高级参数，只有在 Neo4j 专业服务专门提供建议的情况下才能更改
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	5

173	**ha.phase2_timeout**
说　明	Paxos phase 2 的超时时间。如果没有设置，它将默认为 ha.paxos_timeout，如果 ha.paxos_timeout 更改则将隐式更改。这是一个高级参数，只有在 Neo4j 专业服务专门提供建议的情况下才能更改
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	5

174	**ha.pull_batch_size**
说　明	在从服务器上应用（从主服务器获得的）事务的批量大小
有效值	整数
默认值	100

175	**ha.pull_interval**
说　明	从主服务器获取更新的时间间隔
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	0

176	**ha.role_switch_timeout**
说　明	请求线程等待实例成为主服务器或从服务器的超时时间
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	120

177	**ha.server_id**
说　明	集群实例 ID。在集群中必须唯一
有效值	实例 id，必须为有效的整数
强制的	强制性的设置项

178	**ha.slave_lock_timeout**
说　明	在从服务器上进行远程（写）锁的超时时间。默认为 ha.slave_read_timeout
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	20

179	**ha.slave_only**
说　明	该实例在集群中是否仅作为从服务器。如果设置为 true，则不会被选为主服务器
有效值	布尔值
默认值	FALSE

180	**ha.slave_read_timeout**
说　明	从服务器等待主服务器回应的时间长度，超过该时间长度就放弃
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	20

181	**ha.tx_push_factor**
说　明	主服务器将请求复制已提交事务的从服务器数量
有效值	整数，最小值为 0
默认值	1

182	**ha.tx_push_strategy**
说　明	将事务提交给从服务器时的推送策略
有效值	下列之一：round_robin、fixed_descending、fixed_ascending
默认值	fixed_ascending

183	**metrics.bolt.messages.enabled**
说　明	启用 Bolt 协议消息处理的度量报告
有效值	布尔值
默认值	FALSE

184	**metrics.csv.enabled**
说　明	设置为 true，以启用将度量报告导出到 CSV 文件
有效值	布尔值
默认值	FALSE

185	**metrics.csv.interval**
说　明	CSV 文件的报告间隔
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	3

186	**metrics.cypher.replanning.enabled**
说　明	启用有关产生重新计划事件的数目的度量报告
有效值	布尔值
默认值	FALSE

187	**metrics.enabled**
说　明	所有支持的度量的默认启用值。将其设置为 false 则默认关闭所有度量。其后可使用各个设置来选择性地重新启用特定的度量
有效值	布尔值
默认值	FALSE

188	**metrics.graphite.enabled**
说　明	设置为 true 以启用向 Graphite 导出度量报告
有效值	布尔值
默认值	FALSE

189	**metrics.graphite.interval**
说　明	Graphite 的报告间隔时间
有效值	时间长度（有效单位为 ms、s、m；默认单位是 s）
默认值	3

190	**metrics.graphite.server**
说　明	Graphite 服务器的主机名或 IP 地址
有效值	主机名和端口
默认值	:2003

191	**metrics.jvm.buffers.enabled**
说　明	启用关于缓存池的度量报告
有效值	布尔值
默认值	FALSE

192	**metrics.jvm.gc.enabled**
说　明	启用关于垃圾收集期间的度量报告
有效值	布尔值
默认值	FALSE

193	**metrics.jvm.memory.enabled**
说　明	启用有关内存使用情况的度量报告
有效值	布尔值
默认值	FALSE

194	**metrics.jvm.threads.enabled**
说　明	启用关于当前运行线程数量的度量报告
有效值	布尔值
默认值	FALSE

195	**metrics.neo4j.causal_clustering.enabled**
说　明	启用有关因果集群模式的度量报告
有效值	布尔值
默认值	FALSE

196	**metrics.neo4j.checkpointing.enabled**
说　明	启用关于 Neo4j 检查点的度量报告
有效值	布尔值
默认值	FALSE

197	**metrics.neo4j.cluster.enabled**
说　明	启用有关 HA 集群信息的度量报告
有效值	布尔值
默认值	FALSE

198	**metrics.neo4j.counts.enabled**
说　明	启用关于数据库中大约有多少实体（节点、关系、属性等）的度量报告
有效值	布尔值
默认值	FALSE

199	**metrics.neo4j.enabled**
说　明	启用所有 Neo4j 特定支持的度量的默认启用值
有效值	布尔值
默认值	FALSE

200	**metrics.neo4j.logrotation.enabled**
说　明	启用有关 Neo4j 日志轮换的度量报告
有效值	布尔值
默认值	FALSE

201	**metrics.neo4j.network.enabled**
说　明	启用有关网络使用情况的度量报告
有效值	布尔值
默认值	FALSE

202	**metrics.neo4j.pagecache.enabled**
说　明	启用有关 Neo4j 页面缓存的度量报告
有效值	布尔值
默认值	FALSE

203	**metrics.neo4j.server.enabled**
说　明	启用有关服务器线程信息的度量报告
有效值	布尔值
默认值	FALSE

204	**metrics.neo4j.tx.enabled**
说　明	启用关于事务的度量报告
有效值	布尔值
默认值	FALSE

205	**metrics.prefix**
说　明	度量报告字段名称的公用前缀。默认值是“neo4j”，在 HA 配置中运行时，其默认值是一个基于集群和实例名称的计算值
有效值	字符串
默认值	neo4j

206	**tools.consistency_checker.check_graph**
说　明	在节点、关系、属性、类型和标记之间执行检查
有效值	布尔值
默认值	TRUE

207	**tools.consistency_checker.check_indexes**
说　明	执行索引检查。检查索引要比检查原生库的代价更高，因此对于非常大的数据库，建议关闭此检查
有效值	布尔值
默认值	TRUE

208	**Tools.consistency_checker.check_label_scan_store**
说　明	执行标签扫描库检查。检查该库比检查原生库的代价更高，因此对于非常大的数据库，建议关闭此检查
有效值	布尔值
默认值	TRUE

209	**tools.consistency_checker.check_property_owners**
说　明	对属性所有权执行可选的附加检查。这可以检测出“一个属性由多个实体拥有”等理论不一致性。但该检查在时间和内存方面的代价非常高，因此默认为 false（跳过此项检查）
有效值	布尔值
默认值	FALSE

附录B

Neo4j 内建过程

本部分主要描述与 Neo4j 捆绑的内嵌过程，包括两类过程：（1）通用过程；（2）本地用户和角色管理的过程。

通用过程

Neo4j 捆绑了一些内嵌的过程。这些过程可以用于：

（1）检查模式(schema)。
（2）检查元数据。
（3）发现并列出过程和组件。
（4）监控管理数据。

运行 CALL dbms.procedures()将显示所有过程的完整列表。

过程名	调用命令	作用
ListLabels	CALL db.labels()	列出数据库中所有标签（label）
ListRelationshipTypes	CALL db.relationshipTypes()	列出数据库中所有关系类型（Type）
ListPropertyKeys	CALL db.propertyKeys()	列出数据库中所有属性键
ListIndexes	CALL db.indexes()	列出数据库中所有索引
AwaitIndex	CALL db.awaitIndex(label, property, timeout)	等待指定的索引准备就绪
ListConstraints	CALL db.constraints()	列出数据库中所有约束
ListProcedures	CALL dbms.procedures()	列出数据库中所有过程
ListFunctions	CALL dbms.functions()	列出数据库中所有用户函数
ListComponents	CALL dbms.components()	列出数据库的组件及其版本
QueryJmx	CALL dbms.queryJmx(query)	按域及名称查询 JMX 管理器数据，如“org.neo4j:*”
dbms.setTXMetaData	CALL dbms.setTXMetaData(<map>)	向事务附加一个元数据。当列出查询操作时，该数据将被显示，并且会被插入到查询日志里

本地用户和角色管理的过程

下列过程用于本地用户和角色管理，它们只在 Neo4j 企业版中可用。

过程名	说明
dbms.security.activateUser	激活某用户
dbms.security.addRoleToUser	为用户分配一个角色
dbms.security.changeUserPassword	修改用户的密码
dbms.security.changePassword	修改自己的密码
dbms.security.createRole	创建一个自定义角色
dbms.security.createUser	添加一个用户
dbms.security.deleteRole	删除一个自定义角色
dbms.security.deleteUser	删除一个用户
dbms.security.listRoles	显示所有角色
dbms.security.listRolesForUser	显示某用户的所有角色
dbms.security.listUsers	显示所有用户
dbms.security.listUsersForRole	显示某角色的所有用户
dbms.security.removeRoleFromUser	删除某用户的一个角色
dbms.security.suspendUser	暂停一个用户

We have awoken to the fact that relationships are where true value lies in data. Powered by this awareness we have seen the large Web companies - Facebook, Google, LinkedIn, eBay to name but a few - deploy graph technology to devastating advantage.

Dr. Jim Webber

Chief Scientist at Neo4j

如果将大数据应用比作一架高速航行的飞机，数据库就好比飞机的核心动力引擎，那么图数据库就是新一代高效、省油的大推力引擎。

求伯君

WPS创始人

图数据库将在金融、社交、传媒等诸多领域点燃革命性的"星星之火"！

周鸿祎

360公司创始人

图数据库，它是以数学中图论为理论根基，用节点和关系所组成的图为真实世界直观建模，支持百亿乃至千亿量级规模的巨型图的高效关系运算，势必将成为我们这个时代一颗璀璨的明珠！

胡晓峰

国防大学教授、博导

这本书让大家看到先进的科学技术如何在几十年间让冰冷的数据绽放出不亚于音乐和文学的美。

周 涛 博士

电子科技大学

Neo4j 权威指南



图数据库是应"大数据时代"之运而产生的一种新型NoSQL数据库。它以图论为理论根基，用节点和关系所组成的图为真实世界直接建模，支持百亿乃至千亿量级规模的巨型图的高效关系运算，非常适合于高效、便捷、直观地分析和处理大数据中的复杂关系，已得到政商学界的广泛关注。

目前市面上有关该领域的中文书籍非常匮乏，为加大推广和普及的力度，我们集中组织国内对图数据库有深入研究的行业领导者及专家，共同编写一套国人原创的图数据库技术丛书，开篇之作首选全面深入地介绍目前最优秀、市场占有率最高的图数据库产品——Neo4j。全书涵盖基本概念、基本操作和丰富的现实案例，尤为耀眼的是包含对团队原创的Neo4j简体中文版及其扩展功能的介绍。无论是入门还是高级进阶均具有极高的参考价值，大数据、图数据库爱好者更值得拥有！

清华社官方微信号



扫 我 有 惊 喜

ISBN 978-7-302-47776-1



定价：99.00元